# Sonido, interacción y redes

Jordi Duch i Gavaldà
Heliodoro Tejedor Navarro

PID_00188486






*Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es*

# Índice

# Introducción

En este módulo profundizaremos en todos los mecanismos de interacción que intervienen en un videojuego. Hemos dividido este trabajo según la dirección del flujo de información que se da en los diferentes tipos de interacción:

- Usuario a videojuego
- Videojuego a usuario
- Videojuego a videojuego

En la interacción usuario a videojuego, explicaremos cómo usar el teclado, el ratón, el *joystick*, los sensores y los sistemas *multitouch*. Hay otras clases de interacción en este apartado, como el micrófono y los gestos, que no estudiaremos porque se apartan del objetivo del módulo.

La interacción videojuego a usuario es quizás la que más abunda. Empezaremos viendo cómo facilitarle información al usuario y, sobre todo, de qué forma, usando la pantalla (lo que se conoce por usabilidad). Continuaremos explicando maneras de trabajar con el audio: el sonido ambiente, los efectos especiales, los mezcladores... Para finalizar este apartado, hablaremos de los mecanismos de *feedback*, que mayoritariamente suelen ser de fuerza (*force feedback*).

En el último apartado, veremos la interacción videojuego a videojuego, donde repasaremos la API básica de comunicación por red (TCP/IP y *sockets*), diferentes mecanismos de comunicación (TCP, UDP, *peer to peer*) y entraremos en detalle en cómo usar librerías de alto nivel para comunicar varios procesos. Finalmente, explicaremos cuáles son los requisitos para desarrollar un juego en línea persistente.

# Objetivos

En este modulo didáctico, presentamos al alumno los conocimientos que necesita para alcanzar los siguientes objetivos:

1. Entender el proceso de captura de información del entorno y generación de respuestas.

2. Saber diseñar interfaces intuitivas y de fácil uso.

3. Usar con medida los mecanismos de interacción que requiera el videojuego.

4. Conocer los mecanismos necesarios para dotar de juego en red a nuestra aplicación.

# 1. Usuario a videojuego

El primer tipo de sistema de comunicación que vamos a estudiar son los sistemas de entrada de datos (también conocidos como dispositivos de entrada).

Los sistemas de entrada de datos son elementos hardware que transforman información proveniente del mundo real para que pueda ser procesada por un dispositivo electrónico. En el entorno de los videojuegos, estos sistemas permiten que el usuario comunique sus órdenes e intenciones al procesador del juego para poder interaccionar con los elementos que definen nuestro juego.

Para estudiar este punto, primero veremos la historia y clasificación de los diferentes dispositivos de entrada que podemos encontrar en el mercado; explicaremos diferentes formas de captura de esa información y, finalmente, la forma correcta de interpretar esta entrada.

## 1.1. Historia de los dispositivos de entrada

Los sistemas de entrada de datos aparecieron casi en paralelo con las primeras unidades de procesamiento alrededor de 1950 para poder introducir los datos y el código de los algoritmos necesarios para su funcionamiento:

- El **teclado** se adaptó de las máquinas de escribir para obtener el primer sistema de entrada de datos genérico.
- En 1963 se introdujo el **ratón** como el primer dispositivo apuntador, que permitía a los usuarios definir puntos sobre un espacio.

Estos dos elementos combinados (teclado y ratón) se han convertido en el principal conjunto de entrada de datos de la historia de los ordenadores.

En el campo de los videojuegos, los dispositivos de entrada son más conocidos como **controladores de videojuegos** o simplemente controladores.

El primer controlador específico para un videojuego fue creado por los mismos que diseñaron el primer videojuego, Spacewars, para poder controlar las naves. Debido a la dificultad de jugar con los teclados de la época y las ventajas que

***Joystick***

Anteriormente, la palabra *joystick* se había utilizado en aeronáutica para referirse a algunas palancas auxiliares de los aviones.

tenía un jugador sobre el otro cuando se usaba un teclado, crearon un sistema basado en *sticks* para controlar la dirección de las naves, creando el primer *joystick* de la historia.

El desarrollo de controladores se incrementó con la aparición de las máquinas recreativas. Para poder facilitar la interacción con los primeros juegos que salieron al mercado, compañías como Atari crearon sistemas muy simples basados en *joysticks* y botones que permitían el acceso a todo tipo de juegos para todo tipo de usuarios. Con la aparición de las primeras consolas caseras, Atari importó la idea del *joystick* de sus recreativas para crear su Joystick Classic Atari, un controlador que sirvió como referencia para un gran número de controladores posteriores.




Figura 1. Joystick Atari 2600 © Atari

Algunos años más tarde apareció otro controlador que marcaría un estándar dentro de la evolución de los videojuegos, el D-pad (abreviatura de *directional pad*). Fue sobre todo impulsado por Nintendo en sus máquinas portátiles Game & Watch (como Donkey Kong) y en su primera videoconsola personal, la NES/Famitsu. Se trataba de un cursor en forma de cruz en la izquierda que permitía controlar la dirección de los personajes, y de dos botones en la derecha para

activar las acciones del personaje (disparar, saltar...). Debido a su facilidad de uso, el mando se extendió a otras plataformas de otras compañías, y estableció la base de los actuales controladores de las videoconsolas.




Figura 2. Controlador D-pad de la NES © Nintendo

Las siguientes generaciones de controladores fueron evolucionando añadiendo más botones y más direcciones al D-pad, introduciendo uno o dos *sticks* analógicos o, incluso, añadiendo las funcionalidades de vibración.

La diferencia entre los *joysticks* originales y los analógicos se presenta en su arquitectura. Los *joysticks* originales tienen un diseño con cuatro microrruptores y sólo pueden indicar si el mando está arriba, abajo, izquierda o derecha; en cambio, los *joysticks* analógicos tienen un sensor que detecta el grado de inclinación del mando sobre dos ejes (por ejemplo, usando potenciómetros).

El principal objetivo de todos estos añadidos era proporcionar al usuario más posibilidades de interactuar con la máquina (por ejemplo, en los juegos de fútbol se añadían nuevos tipos de pase a medida que se incorporaban botones) o también la posibilidad de realizar una misma acción de diferentes maneras.

Dentro de este grupo de mandos evolucionados, la referencia la estableció Sony con la introducción del PlayStation Dual Shock Controller. Este controlador incorporaba todas las funcionalidades anteriores, pero con un diseño muy ergonómico y que se ha mantenido durante los últimos diez años.

**Cambio de forma**

Sony intentó cambiar la forma del mando para su Playstation 3, pero debido a la presión popular, al final decidió mantener el mismo formato y evitar así a los jugadores tener que acostumbrarse a una nueva forma.




Figura 3. Controlador Playstation 2 Dual Shock © Sony

Paralelamente a la evolución de los mandos, fueron apareciendo otros controladores para juegos específicos (volantes para juegos de carreras, alfombras para juegos de baile, micrófonos para juegos de canto) o con objetivos más generales, como la cámara EyeToy, que se podía integrar junto con otros controladores. Y en algunos casos se trataba de controladores específicos para un solo juego, como los bongos para el Donkey Kong mencionados en el módulo de "Introducción a los videojuegos".

Aunque la última gran revolución en los controladores se produjo con la última generación de consolas, mediante la introducción de controladores inalámbricos con sensores de movimiento y aceleración que ofrecen nuevas formas de interactividad consola-usuario. El ejemplo paradigmático se encuentra en la Nintendo Wii. Esta consola fue la primera que se diseñó alrededor de las mejoras de su controlador (y no mejorando su capacidad de proceso) que ofrecía múltiples nuevas formas de interacción.

El resto de compañías, debido al éxito de esta consola, han rediseñado recientemente la estrategia de sus sistemas para dar más importancia a los sistemas de interacción. Así, Microsoft ha hecho una fuerte apuesta por Kinect, mientras que Sony ha hecho lo mismo por Move. Ambas compañías, junto con

Nintendo, han asegurado que las novedades de las próximas generaciones de consolas estarán muy ligadas a la interacción con el usuario (gestos, voz, sensores, controladores táctiles, ...).

Finalmente, otro campo que ha apostado por la evolución de los sistemas de interacción es el de las máquinas recreativas, ya que a nivel de gráficos estas se encuentran al mismo nivel que las consolas. Por ejemplo, podemos encontrar simulaciones de elementos que serían muy difíciles de trasladar a un comedor: sistemas de ski con plataformas de movimiento, simuladores de motos o coches, etc.

### 1.2. Clasificación de los tipos de entrada

Como hemos comentado, los dispositivos de entrada normalmente están compuestos de diferentes elementos para poder ofrecer una interacción más compleja.

Podemos clasificar los elementos básicos según varios criterios:

- Si proporcionan información discreta o continua.

> Aunque en un sistema digital como son las computadoras no es posible trabajar con información continua, hay sensores de entrada que dan información en un tiempo de respuesta mínimo y con una resolución muy alta que, para la mayoría de aplicaciones, se puede considerar continua (por ejemplo, el uso de un ratón o un sensor de inclinación).

- Si proporcionan información relativa o absoluta.
- El número de grados de libertad que proporcionan (normalmente entre una y tres dimensiones).
- El formato de entrada de la información (verdadero/falso, coordenadas, escalares, ondas...).

Los elementos básicos que componen los sistemas de entrada son los siguientes:

- **Pulsadores**. Proporcionan información sobre el estado de un elemento, devolviendo un valor de verdadero o falso para saber si el botón está pulsado o no en un momento determinado. Los principales elementos basados en pulsadores son las teclas y los botones de los *gamepads*, aunque también podemos encontrar otros sistemas de interacción basados en pulsadores como por ejemplo una guitarra o una alfombra de baile.




Figura 4. Controlador del juego Dance Dance Revolution para Xbox 360 © Konami

- **Sistemas de coordenadas**. Informa de una posición en la pantalla en forma de coordenadas, normalmente en 2 dimensiones (X e Y). Podemos encontrar elementos como el clásico ratón, que proporciona información relativa a una posición, aunque también se pueden utilizar otros dispositivos como una pantalla táctil (útil para realizar videojuegos para móviles o PDA) o los punteros (por ejemplo las pistolas), que proporcionan las coordenadas absolutas donde el usuario quiere interactuar.

- ***Sticks***. Proporcionan el movimiento y/o inclinación en un eje. A diferencia del ratón, no da un valor de posición, sino el grado que existe entre el eje del *stick* y la base del dispositivo. El caso más típico es el del *joystick*, que da información de la inclinación en 2 ejes.

- **Otros sensores**. La industria electrónica ha perfeccionado muchos dispositivos que se están aplicando a la informática para crear nuevas formas de interacción. Cada vez hay más dispositivos que incorporan un acelerómetro (portátiles y teléfonos móviles, por ejemplo), que es un dispositivo que mide la aceleración en dos o tres ejes. En el terreno de las consolas, hemos comentado que el más innovador es el mando de la Nintendo Wii, ya que ofrece posición e inclinación.

**Acelerómetro**

Con un acelerómetro podemos ser capaces de calcular la inclinación respecto al suelo aprovechando la lectura de los dos ejes y sabiendo que la fuerza de la gravedad actúa sobre él.

- **Nuevos usos de dispositivos existentes**. A partir de dispositivos que ya se utilizan cotidianamente en la informática, como micrófonos y cámaras, se obtiene otro tipo de información con las que el usuario puede interactuar. Por ejemplo, el usuario puede mover un avatar gesticulando delante de una cámara, jugar a un karaoke o hablar con otros jugadores. El problema de este tipo de entrada es que requiere mucho más preprocesamiento por parte de los desarrolladores del juego para poder extraer la información necesaria que nos interesa dentro de estas señales.

Como hemos comentado, lo más normal es que los dispositivos de entrada sean combinaciones de los anteriores. Podemos encontrar desde un ratón que está compuesto de dos pulsadores con un sistema para ubicar el cursor, hasta los actuales controladores de la PS3, compuestos de varios botones, *sticks* analógicos y sensores de inclinación. Además, un sistema que nos permita jugar puede utilizar uno o varios dispositivos de entrada simultáneamente, como es el caso de un PC, donde normalmente se combina la entrada del teclado y del ratón.

**Nintendo Wii**

El control remoto de la Nintendo Wii es un ejemplo de la integración de diferentes tecnologías de captura de datos en un único dispositivo. Ofrece los siguientes controles: un D-Pad, varios botones, sensores de aceleración en tres ejes, posicionamiento tridimensional respecto al monitor, además de proporcionar *feedback* en forma de sonido y vibración.

Al dispositivo principal se le puede conectar otros complementos para mejorar la interacción del usuario:

- Por un lado, tenemos un dispositivo llamado Nunchuk, que añade un *stick* analógico, más botones y sensores de aceleración en tres ejes para la otra mano.

- Por otro lado, podemos añadir un mando adicional más clásico, que incluye, un D-Pad, varios botones y dos *sticks* analógicos.

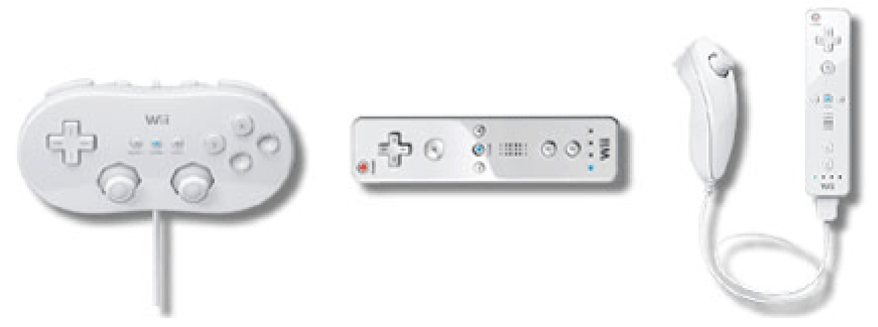



Figura 5. Nintendo Wii Classic Controller, Nintendo Wiimote, Nintendo Nunchaku © Nintendo

El trabajo del programador es interpretar estas señales de entrada para darles un significado en el mundo del videojuego.

Cuando tengamos que leer las órdenes provenientes del mundo exterior, deberemos:

- Leer toda la información que provenga de los componentes de todos los dispositivos de entrada que el usuario pueda utilizar.
- Interpretar las diferentes entradas para deducir qué acciones ha decidido realizar el usuario.

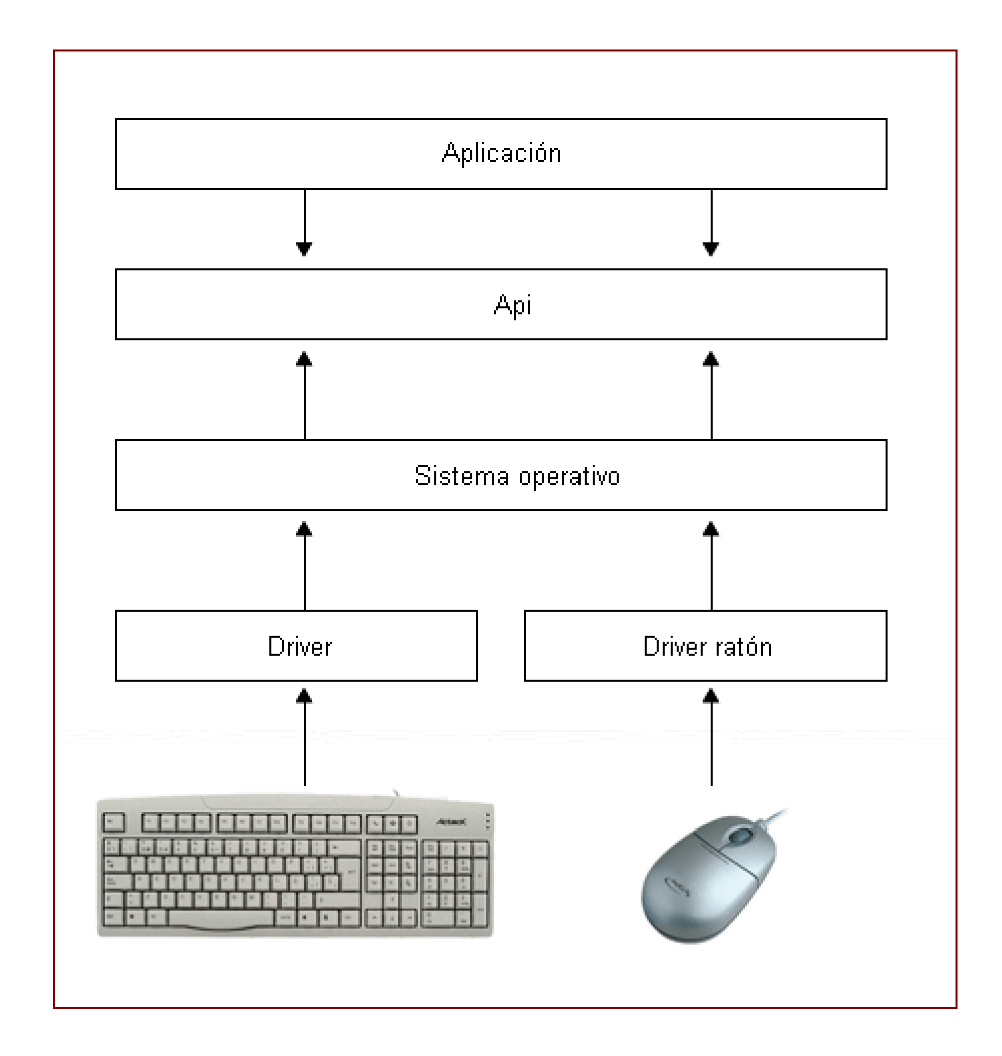
A continuación vamos a estudiar estos puntos por separado.

### 1.3. Lectura de datos de los dispositivos de entrada

En primer lugar, tenemos que entender cómo se transmite, se recibe y se procesa una acción del usuario sobre cualquier componente, por parte de la aplicación que está controlando el juego. Más tarde veremos una pequeña comparativa entre los dispositivos de entrada que podemos tener en un PC y en consolas. Finalmente, estudiaremos las diferentes propuestas que tenemos en el mercado para facilitar las lecturas de los dispositivos.

#### 1.3.1. Funcionamiento de los dispositivos

La clave para facilitar la compatibilidad de un juego con múltiples dispositivos y plataformas / sistemas operativos diferentes es la organización en capas que abstraen lo que está por debajo de ellas:




Hay dos formas básicas para leer la información de un dispositivo:

- **Por interrupción**. El dispositivo detecta un cambio de estado (por ejemplo, el usuario pulsa un botón) y es el responsable de enviar esta información a la unidad central.

- **Por encuesta**. La unidad central pregunta constantemente al dispositivo por su estado; el dispositivo simplemente responde según el protocolo establecido (estado del botón, posición del *stick*...).

La información la recibe un proceso que se ejecuta en la unidad central que llamamos *device driver*. Un *driver* es un componente software que permite que el sistema operativo interactúe con el hardware de un periférico.

***Device driver***

A un *device driver* podemos llamarlo "controlador de dispositivo" o "controlador", pero evitaremos traducirlo para evitar confusiones con el "controlador del juego".

Las características más importantes de un *driver* son:

- Cada elemento hardware necesita su propio *driver* especifico para que se puedan interpretar todas las señales que se originen en el controlador.

- Su trabajo es abstraer todas estas señales recibidas e intentar presentarlas de un modo estándar al sistema operativo, facilitando el acceso al hardware a cualquier aplicación.

Si queremos que el juego sea compatible con diferentes sistemas operativos, necesitamos otro elemento que nos permita comunicar nuestro programa con el sistema operativo. Como hemos introducido en los anteriores apartados, este paso lo realizamos mediante el uso de librerías que nos abstraigan la comunicación con el *driver*.

### 1.3.2. Ventajas e inconvenientes de la captura de datos en PC y en consolas

En entornos PC tenemos mucha más versatilidad para escoger los periféricos que queramos para nuestro videojuego debido al mayor número de dispositivos existentes. En cambio en las consolas, el número de controladores es mucho más pequeño, lo que limita la manera de interactuar con el juego.

Como comentamos anteriormente, en este entorno tenemos la posibilidad de desarrollar nuestra propia API como intermediaria entre el juego y el sistema operativo. En el caso de las videoconsolas, en alguna ocasión puntual se desarrolla un nuevo dispositivo de entrada con sus propios controladores. El problema es que estos dispositivos son utilizados por muy pocos juegos, ya que las API necesarias para acceder a ellos son propietarias.

**Ved también**

Podéis consultar el módulo sobre las ventajas e inconvenientes de desarrollar nuestra propia API.

En cambio, en los PC tenemos que preocuparnos de qué alternativas tiene el usuario para realizar una acción cuando al controlador le falta algún botón necesario. Este problema no se reproduce en las consolas debido a que todos los mandos son iguales.

### 1.3.3. Estudio de librerías

El trabajo del programador de videojuegos empieza normalmente a partir de los datos proporcionados por la librería que utilicemos para gestionar la entrada. Algunas librerías disponibles para ordenadores personales más utilizadas en la programación de videojuegos son las siguientes:

- **DirectInput**. Se trata de uno de los componentes de las librerías DirectX, el conjunto de librerías de Microsoft para facilitar el desarrollo de juegos y aplicaciones multimedia para entornos Windows. Soporta un gran número de dispositivos diferentes: *joysticks*, volantes, *pads*, teclados, ratones... Se integra perfectamente en el entorno Windows, permitiendo gestionar eventos siguiendo la filosofía del sistema operativo. Además, incorpora funciones adicionales como la gestión del Force Feedback. El principal problema es que la portabilidad de las aplicaciones desarrolladas con esta API es muy baja (sólo disponibles para Windows y Xbox).

- **SDL_Event**. Se trata del componente para gestionar la entrada de datos de las librerías SDL (Simple DirectMedia Layer). A diferencia del DirectInput, se trata de librerías multiplataforma, lo que facilita la portabilidad entre diferentes sistemas operativos (Linux, Windows, Amiga, Mac OS). Por otro lado, proporciona menos funciones que las DirectX, ya que sólo incluye un método para tratar entrada de datos.

- **Object Oriented Input System (OIS)**. Esta librería la desarrolla Phillip Castaneda y puede descargarse de la web del proyecto Wrecked Game Engine (dedicado a la creación de un motor genérico para videojuegos). Esta librería se incorporó al desarrollo de Ogre a partir de la versión 1.4. Las principales ventajas de esta API son, por un lado, que se trata de un sistema multiplataforma (existen versiones para varios sistemas operativos) y por el otro, que está diseñada para que sea muy modular, lo que la hace fácil de utilizar para principiantes. En contra podemos afirmar que, al tratarse de un proyecto relativamente pequeño, hay muy poco soporte por parte de los desarrolladores de la librería, con lo que limita su posible uso en aplicaciones profesionales.

**Identificador de la ventana**

La librería necesita un identificador de la ventana que capturará los datos y este valor dependerá del sistema operativo. Es el único caso donde debemos tener en cuenta el sistema operativo para el que trabajamos.

El acceso a las librerías de captura de datos en videoconsolas es una tarea más complicada que cuando hablamos de PC. Algunas compañías sólo permiten el acceso a los dispositivos de entrada mediante las librerías propias que se encuentran en los kits de desarrollo (Nintendo), mientras que otras compañías proporcionan las librerías de forma pública y se pueden descargar desde su página web, potenciando el desarrollo de juegos independientes para sus plataformas (Sony o Microsoft). En el caso de no tener acceso a las librerías oficiales, una manera alternativa es utilizar lo que se conoce por librerías *homebrew* (librerías que han sido desarrolladas por los usuarios a partir de técnicas de ingeniera inversa).

En este módulo utilizaremos la librería OIS, que nos permite un acceso muy limpio a los dispositivos de entrada. Para trabajar con la librería, debemos descargarla e incluir el directorio de cabeceras y librerías en nuestro entorno de desarrollo.

En nuestro código, debemos crear una instancia al controlador de entradas (InputManager). A esta instancia le tenemos que pasar el identificador de la ventana que va a capturar los datos. Esta información nos la debe proporcionar el sistema operativo donde trabajamos. Por ejemplo, en Windows quedaría:

**Entrada.h**

```
#ifndef __Entrada__
#define __Entrada__

#include <OIS/OIS.h>
#include <windows.h>

class Entrada
{
public:
   Entrada();
   void init(HWND hWnd);
private:
   OIS::InputManager* m_manager;
};

#endif // __Entrada__
```

**Entrada.cpp**

```
Entrada::Entrada()
   : m_manager(NULL)
{
}

void Entrada::init(HWND hWnd)
{
   OIS::ParamList paramList;
   std::ostringstream os;
   os << (unsigned int) hWnd;
   paramList.insert(std::make_pair(std::string("WINDOW"), os.str()));
   m_manager = OIS::InputManager::createInputSystem(paramList);
}
```

### 1.4. Captura de la entrada

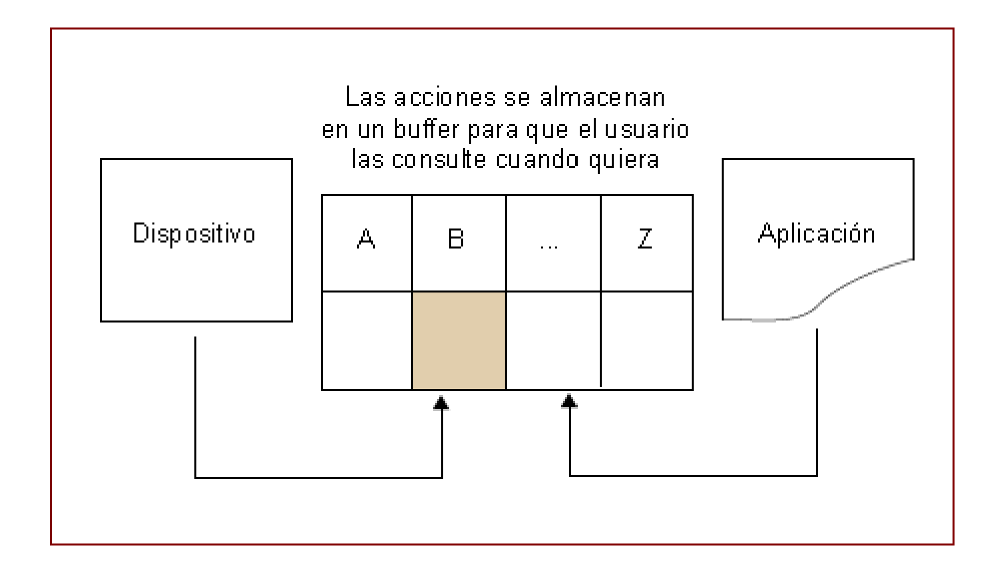
Una vez hemos decidido la API que nos va a proporcionar el estado de los dispositivos de captura de datos, el siguiente paso es leer esta información y proporcionarla a nuestra aplicación.

Hay dos sistemas principales para tratar el estado de los dispositivos de entrada: de forma pasiva (lo que se conoce como sistemas *buffered*) o de forma activa (o sistemas *unbuffered*), que se corresponden con los métodos interrupción/encuesta explicados en la sección anterior. Vamos a estudiar el funcionamiento de ambos sistemas.

#### 1.4.1. Sistemas *buffered*

En este tipo de sistemas, cada vez que queramos saber si una tecla (o cualquier otro evento de entrada) está pulsada o no, tenemos que consultarlo a la API.

La API se encarga de mantener un *buffer* con el estado de todos los elementos que está controlando, como por ejemplo una posición de memoria con un valor booleano para saber qué teclas están pulsadas a la vez. Cuando queremos saber si el usuario está realizando una acción, tenemos que leer la posición correspondiente en el *buffer* y actuar según lo leído.




La principal ventaja de este tipo de sistemas es que no tenemos que preocuparnos de qué está haciendo el usuario cuando no queremos leer información; es decir, esta información no llegará a nuestro programa hasta que nosotros queramos. En cambio, el problema principal es que no recibimos la entrada en el mismo momento que el usuario acciona algún tipo de entrada, creando un pequeño retraso entre la acción y la lectura que puede ser clave en ciertos tipos de juegos.

En el caso de la librería OIS, el código para conectarse al teclado usando un *buffer* sería:

```
void Entrada::createKeyboardBuffered()
{
    if (m_manager->getNumberOfDevices(OIS::OISKeyboard) > 0)
        m_keyboard = (OIS::Keyboard*) m_manager->createInputObject(
          OIS::OISKeyboard, true);
    else
       m_keyboard = NULL;

}
```

Y para leer información sobre el teclado:

```
if (m_keyboard->isKeyDown(OIS::KC_ESCAPE))
   std::cout << "ESCAPE pressed!" << std::endl;
```

El código necesario para utilizar un ratón y un *joystick* sería:

```
void Entrada::createMouseBuffered()
{
    if (m_manager->getNumberOfDevices(OIS::OISMouse) > 0)
        m_mouse = (OIS::Mouse*) m_manager->createInputObject(
            OIS::OISMouse, true);
    else
        m_mouse = NULL;
}

void Entrada::createJoyStickBuffered()
{
    if (m_manager->getNumberOfDevices(OIS::OISJoyStick) > 0)
        m_joyStick = (OIS:: OISJoyStick *) m_manager->createInputObject(
            OIS:: OISJoyStick, true);
    else
        m_ joyStick = NULL;

}
```

Para leer la información del ratón:

```
OIS::MouseState& state = m_mouse->getMouseState();
std::cout << "Coordenada absoluta: " << state.X.abs << ", " << state.Y.abs << std::endl;
std::cout << "Coordenada relativa: " << state.X.rel << ", " << state.Y.rel << std::endl;
```

Aunque para que el ratón nos dé la posición dentro de la ventana debemos indicarle al controlador su tamaño, para ello tenemos que modificar los atributos "width" y "height":

```
OIS::MouseState& state = m_mouse->getMouseState();
state.width = WINDOW_WIDTH;
state.height = WINDOW_HEIGHT;
```

Para leer el estado de los botones del ratón:

```
OIS::MouseState& state = m_mouse->getMouseState();
if (state.buttonDown(OIS::MB_Left))
   std::cout << "Left pressed" << std::endl;
else
   std::cout << "Left released" << std::endl;
```

Y en el caso de leer del *joystick*:

```
OIS::JoyStickState& state = m_joyStick->getJoyStickState();
std::cout << "Ejes X: " << state.mAxes[0].abs << ", y: " << state.mAxes[1].abs << std::endl;
if (state.buttonDown(0))
   std::cout << "First button pressed" << std::endl;
else
   std::cout << "First button released" << std::endl;
```

En el caso de la lectura de los ejes del *joystick*, la lectura siempre será en posiciones absolutas porque nos indica la posición de la palanca, y no como el ratón, que nos indica la posición del apuntador o la distancia recorrida desde la última lectura.

En el método "ButtonDown" del estado del *joystick* debemos pasar como parámetro un valor entero que indica el número de botón. En el caso del *joystick* no existe una relación entre la posición del botón dentro del mando y un nombre que lo defina.

Podemos saber si en un eje se nos permite hacer la lectura en posiciones relativas (movimiento entre la posición anterior y la actual) leyendo el atributo "absOnly" que contiene cada eje:

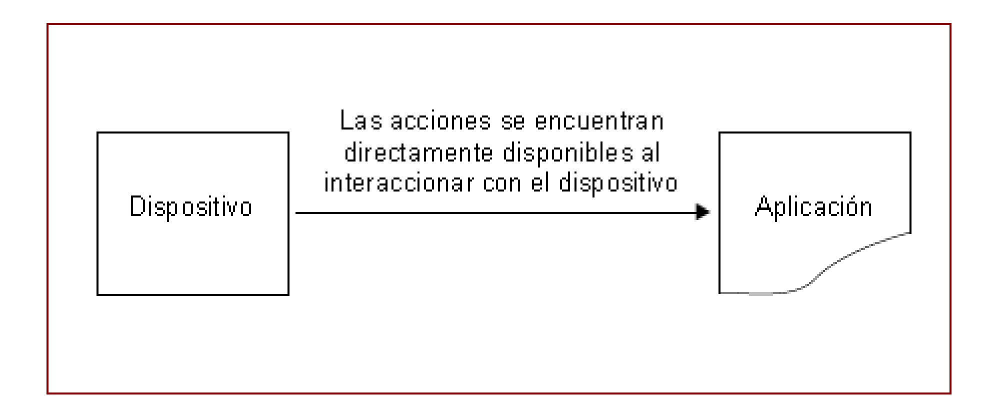
```
OIS::JoyStickState& state = m_joyStick->getJoyStickState();
OIS::Axis& X = state.mAxes[0];
if (! X.absOnly)
   std::cout << "Tambien puedo leer relativo?: " << X.rel << std::endl;
```

Y para leer diferentes elementos que puede integrar un *joystick*, tenemos más atributos, como palancas (en inglés *sliders*) y botones direccionales (en inglés, *pov hat*):

```
OIS::JoyStickState& state = m_joyStick->getJoyStickState();
switch(state.mPOV[0].direction) // POV One
{
case OIS::Pov::Centered:
   std::cout << "Pov centered" << std::endl;
   break;
case OIS::Pov::North:
case OIS::Pov::South:
case OIS::Pov::East:
case OIS::Pov::West:
   std::cout << "Pov horz or vert" << std::endl;
   break;
case OIS::Pov::NorthEast:
case OIS::Pov::NorthWest:
case OIS::Pov::SouthEast:
case OIS::Pov::SouthWest:
   std::cout << "Pov in diagonal" << std::endl;
   break;
}
```

### 1.4.2. Sistemas *unbuffered*

En el caso de un sistema *unbuffered*, cada vez que el usuario interacciona con un dispositivo de entrada, se genera un evento en la API. Este evento es enviado directamente a la aplicación en el mismo momento en que se recibe, y no se almacena en ningún sistema intermedio.



Para gestionar este tipo de eventos, se utiliza lo que se conoce por una función de *callback*. Al principio de la aplicación se registra a qué función se enviarán los eventos de entrada que reciba la API. El trabajo del programador consiste en implementar estas funciones para que sean capaces de recoger toda la información y no perder detalle de lo que hace el usuario.

A diferencia del sistema anterior, al no existir un *buffer* intermedio, no es posible saber en todo momento cuál es el estado de un dispositivo, es decir, no podemos preguntar si una tecla está pulsada en un determinado momento. En el caso de que nos interese llevar el control de lo que está pulsado en cada momento, lo que podemos hacer es mantener nosotros mismos unas variables (o un *buffer*) de control de estado, y actualizarlas dentro de las funciones de *callback* cuando vayan llegando eventos del tipo tecla pulsada / tecla dejada de pulsar.

La principal ventaja de este tipo de sistemas es que reducen al mínimo el tiempo de respuesta de la aplicación. En algún tipo de juegos, como los First Person Shooters, el tiempo de respuesta es un elemento muy importante para evitar la frustración de los usuarios.

**Lag**

Es común utilizar el vocablo inglés *lag* para referirnos al tiempo de respuesta a un evento.

Siguiendo con el ejemplo del teclado, el código usando OIS sería:

**Modificación en entrada.h**

```
...
class Entrada
   : public OIS::KeyListener
{
...
// OIS::KeyListener members
protected:
   bool keyPressed(const OIS::KeyEvent& arg);
   bool keyReleased(const OIS::KeyEvent& arg);
...
```

**Modificación en entrada.cpp**

```
void Entrada:createKeyboardUnbuffered()
{
   m_keyboard = (OIS::Keyboard*) m_manager->createInputObject(
                 OIS::OISKeyboard, false);
   m_keyboard->setEventCallback(this);
}

bool Entrada::keyPressed(const OIS::KeyEvent& arg)
{
   std::cout << "La tecla " << arg.key << " se ha pulsado" << std::endl;
```

```
    return true;
}

bool Entrada::keyReleased(const OIS::KeyEvent& arg)
{
    std::cout << "La tecla " << arg.key << " se ha soltado" << std::endl;
    return true;
}
```

En el caso del ratón y el *joystick*, deberíamos implementar las interfaces OIS::MouseListener y OIS::JoyStickListener que nos permiten saber:

- Cuándo se mueve el ratón por encima de nuestra ventana.
- Cuándo se pulsa o se suelta un botón del ratón.
- Cuándo se ha movido la palanca del *joystick*.
- Cuándo se han pulsado los botones del *joystick*.

**Modificación en entrada.h**

```
...
class Entrada
    : public OIS::MouseListener
    , public OIS::JoyStickListener
{
...
// OIS::MouseListener members
protected:
    bool mouseMoved(const OIS::MouseEvent& arg);
    bool mousePressed(const OIS::MouseEvent& arg, OIS::MouseButtonID id);
    bool mouseReleased(const OIS::MouseEvent& arg, OIS::MouseButtonID id);

// OIS::JoyStickListener members
protected:
    bool buttonPressed(const JoyStickEvent& arg, int button);
    bool buttonReleased(const JoyStickEvent& arg, int button);
    bool axisMoved(const JoyStickEvent& arg, int axis);
    bool sliderMoved(const JoyStickEvent& arg, int index);
    bool povMoved(const JoyStickEvent& arg, int index);
};
...
```

**Modificación en entrada.cpp**

```
void Entrada:createMouseUnbuffered()
{
    m_mouse = (OIS::Mouse*) m_manager->createInputObject(OIS::OISMouse, false);
    m_mouse->setEventCallback(this);
```

```
}

void Entrada::createJoyStickUnbuffered()
{
   m_joystick = (OIS::JoyStick*) m_manager->createInputObject(
                 OIS::OISJoyStick, false);
   m_joystick->setEventCallback(this);
}

bool Entrada::mouseMoved(const OIS::MouseEvent& arg)
{
   std::cout << "Mouse moved to " << arg.state.X.abs << ", " << arg.state.Y.abs
      << std::endl;
   return true;
}

bool Entrada::mousePressed(const OIS::MouseEvent& arg, OIS::MouseButtonID id)
{
   if (id == OIS::MB_Left)
      std::cout << "Left button pressed" << std::endl;
   else
      std::cout << "Other button pressed" << std::endl;
   return true;
}

bool Entrada::mouseReleased(const OIS::MouseEvent& arg, OIS::MouseButtonID id)
{
   if (id == OIS::MB_Left)
      std::cout << "Left button released" << std::endl;
   else
      std::cout << "Other button released" << std::endl;
   return true;
}

bool Entrada::buttonPressed(const JoyStickEvent& arg, int button)
{
   std::cout << "Button #" << button << " pressed" << std::endl;
   return true;
}

bool Entrada::buttonReleased(const JoyStickEvent& arg, int button)
{
   std::cout << "Button #" << button << " released" << std::endl;
   return true;
}

bool Entrada::axisMoved(const JoyStickEvent& arg, int axis)
```

```
{
   const OIS::Axis& a = arg.state.mAxes[axis];
   std::cout << "Axis #" << axis << " moved to " << a.X.abs << ", " << a.Y.abs
      << std::endl;
   return true;
}

bool Entrada::sliderMoved(const JoyStickEvent& arg, int index)
{
   std::cout << "Slider #" << index << " moved" << std::endl;
   return true;
}

bool Entrada::povMoved(const JoyStickEvent& arg, int index)
{
   std::cout << "Pov #" << index << " moved at ";

   switch(arg.state.mPOV[index].direction)
   {
   case OIS::Pov::Centered:
      std::cout << "centered";
      break;
   case OIS::Pov::North:
      std::cout << "north;
      break;
   case OIS::Pov::South:
      std::cout << "south";
      break;
   case OIS::Pov::East:
      std::cout << "east";
      break;
   case OIS::Pov::West:
      std::cout << "west";
      break;
   case OIS::Pov::NorthEast:
      std::cout << "north east";
      break;
   case OIS::Pov::NorthWest:
      std::cout << "north west";
      break;
   case OIS::Pov::SouthEast:
      std::cout << "south east";
      break;
   case OIS::Pov::SouthWest:
      std::cout << "south west";
      break;
   }
```

```
        std::cout << std::endl;
        return true;
    }
```

## 1.5. Interpretación de la entrada

Tras saber qué acción está realizando un usuario en un determinado momento de un juego, la fase final de la captura de entrada es darle un significado a la interacción del usuario; es decir, transformar la entrada de datos en acciones reales dentro del juego.

Es importante recordar que una misma acción se puede efectuar de diferentes maneras y que a veces, para poder realizar una acción, tenemos que asegurarnos de que todos los requisitos se cumplan (por ejemplo, que se pulsen todas las combinaciones de botones y en el orden correcto). En este caso, necesitaremos variables auxiliares que nos permitan registrar las secuencias de eventos del usuario, y realizar la acción si una de estas secuencias se completa.

***Cheat codes***

Uno de los casos más curiosos de secuencias de comandos se encuentra en los conocidos *cheat codes*, que nos permiten desbloquear trucos y opciones especiales mediante la pulsación de largas secuencias de elementos. Los *cheat codes* son parte de las herramientas que los programadores utilizan a lo largo del desarrollo del juego para poder acceder más fácilmente a determinadas zonas y poder experimentar el efecto de modificaciones puntuales.

Otro problema que tenemos que tener en cuenta es que algunos dispositivos pueden cambiar de un país a otro. El problema más destacado es el teclado, pues en cada país varían las teclas, sobre todo por lo que se refiere a teclas especiales y de control, y su disposición en el teclado. Para evitar este problema, las API nos proporcionan una abstracción adicional de los códigos de las teclas, a las que nos referiremos a partir de constantes definidas en la explicación de la API.

En el caso del OIS, un ejemplo de las constantes más utilizadas son las siguientes:

| Constante de OIS | Tecla |
|---|---|
| KC_ESCAPE | Escape |
| KC_LEFT | Cursor izquierda |
| KC_RIGHT | Cursor derecha |
| KC_UP | Cursor arriba |
| KC_DOWN | Cursor abajo |
| KC_LCONTROL | Tecla control (de la izquierda) |
| KC_RCONTROL | Tecla control (de la derecha) |

| Constante de OIS | Tecla |
|---|---|
| **KC_LSHIFT** | Tecla mayúsculas (de la izquierda) |
| **KC_RSHIFT** | Tecla mayúsculas (de la derecha) |
| **KC_RETURN** | *Return* |
| **KC_SPACE** | Barra espaciadora |
| **KC_HOME** | Tecla Inicio |
| **KC_END** | Tecla fin |
| **KC_PGUP** | Tecla página arriba |
| **KC_PGDOWN** | Tecla página abajo |

En el caso de tener un *joystick* analógico, su uso puede darnos algún que otro problema si no tenemos precauciones. La palanca del *joystick* apenas nos dará una lectura en el centro (cero), sino que siempre estará oscilando sobre este punto. En un juego en el que el avatar tenga un movimiento discreto (típico juego arcade en el que el avatar puede correr a la izquierda, a la derecha o quedarse quieto), tenemos que decidir a partir de qué valores se mueve y cuándo está quieto. Por ejemplo:

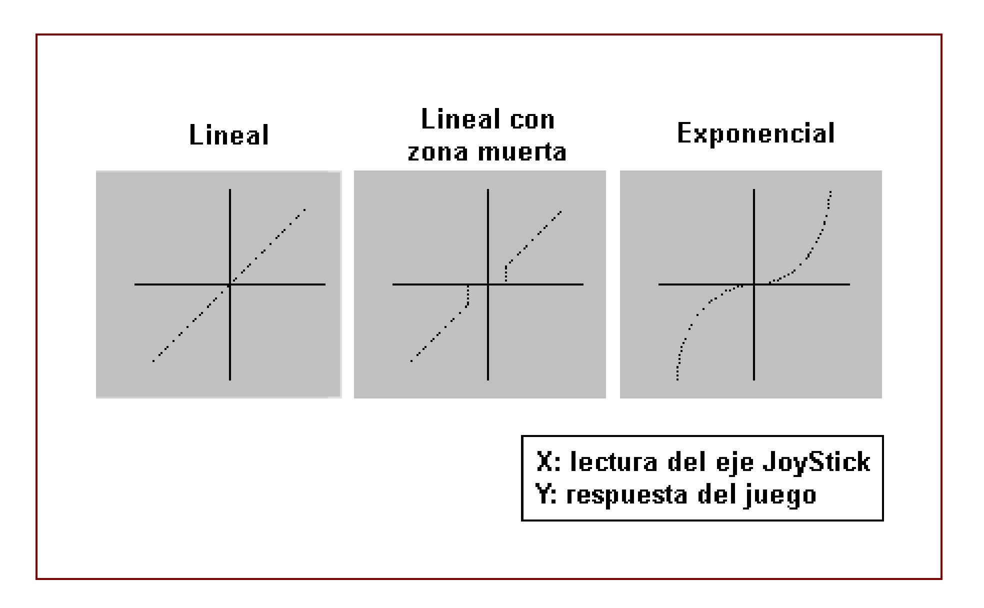
| Valor *joystick* | Acción |
|---|---|
| < **0** | Correr izquierda |
| **0** | Quedarse quieto |
| > **0** | Correr derecha |

Si sabemos que el rango de valores que nos proporciona el *joystick* varía entre -127 y 128, podemos crearnos esta tabla:

| Valor *joystick* | Acción |
|---|---|
| < **27** | Correr izquierda |
| >=**27** & <= **28** | Quedarse quieto |
| > **28** | Correr derecha |

O bien podemos permitir al usuario modificar el valor de la frontera usando una pantalla de calibración (selección de mayor o menor precisión).

También podemos tener este problema si simulamos la dirección de un coche según la posición del *joystick*. En este caso, el coche siempre estará girando si no tratamos la información de entrada. Para mitigar estos efectos, tenemos las siguientes funciones que nos pueden ayudar:




Una vez tenemos toda la información de la entrada, lo siguiente es decidir la acción que hay que realizar por parte del código para responder a la entrada del usuario. La decisión se tiene que tomar en función de dos parámetros:

- El estado en el que se encuentra el juego.
- El contexto específico donde se ha producido la acción.

Volveremos a tratar el problema de la interpretación de la entrada cuando estudiemos el motor lógico de un videojuego, que es el que nos proporcionará la manera de saber qué está pasando en un momento determinado en el juego, y por tanto nos dirá qué tenemos que hacer con la entrada de datos recibida.

### 1.5.1. Selección de objetos

En la interpretación de las señales de los dispositivos de entrada, quizás el problema más complejo es el de la selección de objetos. La selección se realiza mediante dispositivos apuntadores, como ratones o sensores de posición. El usuario indica un punto o una zona de la pantalla, y a partir de aquí tenemos que identificar qué está viendo en esta zona, y por tanto, decidir qué es lo que el usuario ha seleccionado o ha elegido. La selección se utiliza mucho en algunos géneros específicos, como por ejemplo las aventuras gráficas, los juegos de estrategia y en algunos tipos de juegos de rol.

En el caso de juegos 2D, la dificultad de interpretar los objetos seleccionados es menor. Normalmente, recibimos una o dos coordenadas posicionales (x,y), y simplemente hemos de ponerlas en el contexto del juego (es decir, traducirlas a posiciones absolutas) y posteriormente mirar qué elementos entre estas dos posiciones se pueden seleccionar.

El problema es que normalmente dentro del área de selección puede haber distintos tipos de elementos, tales como elementos estáticos que identifican al mundo, elementos que controlamos nosotros (y que por tanto podemos selec-

cionar) y otros elementos que no nos pertenecen. Para decidir qué elementos se incluyen en la selección de una región, se utiliza lo que se conoce por máscaras de selección. Vamos a explicar cómo funciona el proceso de selección con máscaras:

- En primer lugar, se asigna a todos los objetos que existan en el mundo un valor de máscara. Este valor identifica en qué momento podrán ser seleccionados los objetos.

- Cuando el usuario selecciona una región de la pantalla, se obtienen todos los objetos que existan en esta región y se devuelven en un *buffer*.

- Antes de entregar la lista de objetos seleccionados al programa, utilizamos un filtro condicional para decidir qué objetos deben ser seleccionados en este momento a partir de las condiciones del juego.




Figura 6. Warcraft: Orcs and Humans © Blizzard Entertainment. La imagen inferior nos muestra las máscaras de selección de los diferentes elementos.

El problema de la selección se complica cuando el mundo en el que trabajamos es tridimensional. En este caso, la selección de los objetos no se puede efectuar mediante un procedimiento directo, ya que no hay una correspondencia directa entre coordenadas de pantalla y coordenadas del mundo real. Además, en cada punto 2D de la pantalla tenemos representados infinitos puntos en 3D.

La selección 3D se puede realizar mediante OpenGL. Vamos a introducir un método más genérico que se utiliza en otros sistemas, lo que conocemos por un Ray Casting. Explicaremos en qué consiste y cómo se utiliza la técnica del Ray Casting:

- Un Ray Casting consiste en un rayo proyectado desde la posición del objetivo de la cámara y que atraviesa el punto que se corresponde con las coordenadas del ratón.

- Mediante una función de intersección entre línea y objeto, podemos determinar todos los objetos que el rayo va cruzando, y los vamos almacenando en un *buffer* ordenados por la distancia entre el objeto y la cámara.

- Finalmente, al igual que hacemos en 2D, filtramos el resultado del *buffer* para obtener todos aquellos objetos que nos interese seleccionar. Devolvemos al usuario el objeto más cercano que haya quedado en el *buffer*.

- Para hacer una selección de una región en lugar de un punto, no se calcula la intersección de una línea con los objetos del mundo, sino que se hace la intersección con una pirámide y se tienen en cuenta todos aquellos objetos interiores a la pirámide.

La mayoría de motores gráficos incorporan funciones de alto nivel para efectuar esta tarea, lo que facilita mucho el trabajo del programador, ya que únicamente tiene que informar de la posición del ratón y de la máscara de selección que quiera utilizar. Estos motores son capaces de hacer esta tarea de forma óptima, ya que simplemente buscan el primer objeto que cumpla con la máscara de selección, y por tanto evitan el uso de un *buffer* intermedio.

### 1.6. Nuevos sistemas de interacción

Como comentamos en la introducción, la última generación de consolas ha revolucionado la importancia de los sistemas de interacción. Esto se ha conseguido integrando en los mandos convencionales nuevos elementos hardware que permiten complementar los tipos de interacción existentes o introducir nuevos tipos.

Estos elementos podemos agruparlos en cuatro tipos principales:

- Sistemas basados en sonido: usar la voz como controlador.
- Sistemas basados en vídeo: usar el cuerpo como controlador.
- Sistemas basados en sensores: usar sensores (acelerómetros, giroscopios, ...) como controladores.
- Sistemas basados en el tacto: usar el input de los dedos como controlador.

En esta sección vamos a repasar las tecnologías asociadas con el procesamiento de los sensores y los sistemas táctiles. La complejidad de explicar sistemas basados en audio y vídeo se queda un poco fuera del ámbito del libro, pero se puede encontrar en Internet más información sobre el tema. Además, estos

dos tipos de elementos de interacción todavía se encuentran en una fase primaria de su desarrollo e implantación, mientras que los otros dos se encuentran instalados en cualquier *smartphone* actual.

### 1.6.1. Sensores

Los sensores son dispositivos hardware específicos que capturan o miden el estado del controlador. Algunos de los más típicos y que podemos encontrar son los siguientes:

- Acelerómetro: mide la fuerza que se está aplicando al dispositivo sobre cada uno de sus tres ejes. Permite conocer la aceleración o la gravedad.

- Giroscopio: mide la aceleración angular en los tres ejes. Permite controlar la cantidad de giro.

- Magnetómetro: mide los campos magnéticos alrededor del dispositivo. Permite determinar, por ejemplo, dónde se encuentra el norte.

- Infrarrojos: permiten detectar puntos de luz en el espacio y determinar la posición del controlador con respecto a ellos.

Explicar la implementación de cada uno de los sensores es complicado: cada fabricante tiene su propia API con funciones específicas, los datos se codifican en diferentes unidades, ... Aun así, vamos a dar una serie de consejos generales que se aplican a cualquier código que tenga que trabajar con sensores:

- Hay dos formas de recoger la información de los sensores, mediante *polling* o mediante interrupciones. Si optamos por *polling*, desde el código principal consultaremos constantemente el estado del sensor y actuaremos en consecuencia. Si optamos por interrupciones, tendremos que implementar una función de *callback*, que se activará cada vez que se produzca un evento (o cada cierto tiempo si la programamos con un *timer*). Esta función será la que reciba los datos del estado del sensor y los transformará en acciones del usuario. Escoger un método u otro dependerá de las necesidades del juego y de las funciones proporcionadas por la API.

- Se debe conocer bien cómo se codifican los datos y cuáles son los ejes de referencia del sensor para interpretar correctamente la información. Esta información está disponible en la API y en los manuales de los sistemas.

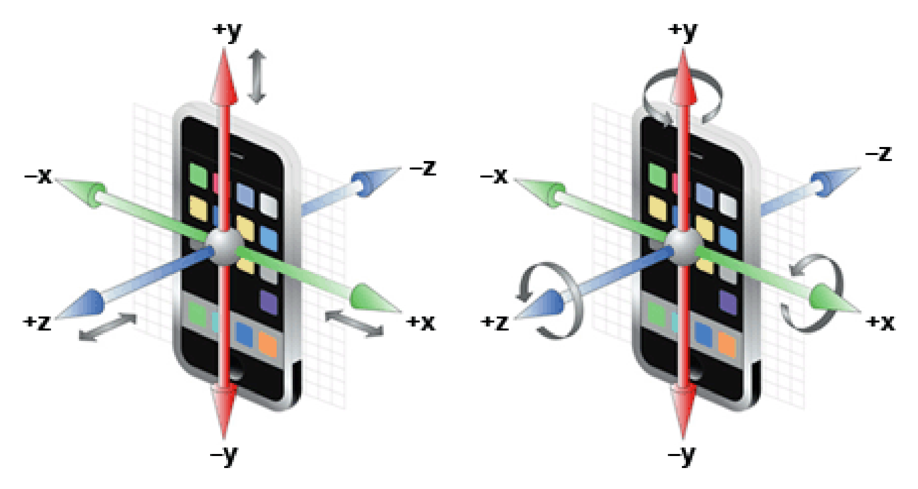



Figura 7. Configuración de los ejes de posicionamiento y de rotación para los smartphones de Apple. © Apple Inc.

- La información que se recibe de los sensores normalmente se encuentra sin tratar, lo que conlleva algunos problemas: tienen muchas fluctuaciones, puede haber errores en las lecturas... Para mejorar la experiencia del usuario, es muy importante tratar estos valores mediante filtros (como, por ejemplo, descartar todo cambio que no supere un límite inferior) o correcciones (como, por ejemplo, eliminar el efecto de la gravedad) para mejorar el control y la sensibilidad del dispositivo.

**Crítica al WiiMote**

Una de las críticas más importantes que recibieron los juegos que usaban el WiiMote de Nintendo era su falta de estabilidad y precisión en capturar el movimiento.

### 1.6.2. Sistemas táctiles

Aunque los sistemas táctiles también son sensores, los tratamos en un punto aparte debido a su importancia en muchos de los dispositivos de juego actuales, como son algunas consolas portátiles y sobre todo los smartphones.

La mayoría de estos dispositivos llevan sistemas multitáctiles, es decir, que permiten capturar la posición y movimiento de varios dedos de forma simultánea (y, en algunos casos, incluso la presión de los dedos en la superficie). Si el sistema está montado encima de una pantalla se le llama pantalla táctil, si no, se le llama *touchpad*.

***Multi-touch***

La palabra *multi-touch* la registró Apple cuando presentó su primer iPhone que incorporaba esta tecnología.

La gran mayoría de sistemas multitáctiles permiten capturar dos tipos de interacciones que se acostumbran a tratar por separado, los toques en la pantalla (*touches*) y los gestos con uno o varios dedos simultáneamente (*gestures*).

En el caso de los touches, las API de los sistemas nos informan de cuándo ponemos un dedo, cuándo lo movemos y cuándo lo levantamos. En el caso de las *gestures*, nos informan de cuatro tipos de eventos diferentes: los toques (*taps*), los desplazamientos (*scrolls/swipes*), las reducciones/ampliaciones (*pinch/spread*) y las rotaciones (*rotations*).

A la hora de programar un juego, estas son algunas recomendaciones a tener en cuenta a la hora de programar sistemas multitáctiles:

- En la gran mayoría de los casos debemos usar la información de los toques directamente. Si nos interesa trabajar con gestos, es mejor que nosotros mismos nos implementemos la detección de los gestos que queramos para tener más control sobre ellos.

- Debemos controlar si la información que se recibe cuando se inicia el *touch* o cuando desplaza el dedo es relativa al origen o si es en coordenadas absolutas. Dependiendo de las necesidades, tendremos que combinarlas y/o transformarlas a las coordenadas de visualización.

# 2. Videojuego a usuario

El ordenador debe de ser capaz de proporcionarnos información sobre el estado del juego. Para ello debemos conocer todos los tipos de dispositivos que podemos utilizar para tal fin.

Como ya conocéis los dispositivos gráficos, los más utilizados para llevar a cabo esta interacción, dejaremos un poco de lado la tecnología y nos centraremos en cómo diseñar las interfaces gráficas.

El sonido es otra forma de presentar información al usuario. Conoceremos cómo funciona el sonido y cómo aprovechar esta tecnología para aportarle sensaciones al jugador.

Finalmente, daremos un breve repaso a las técnicas de *feedback* (sobre todo vibraciones mecánicas) que ofrecen los nuevos controladores de juego.

## 2.1. Pantalla

Para poder hacer cualquier tipo de aplicación, y sobre todo un videojuego, es necesario dedicar mucho interés a la manera como se presenta esta información al usuario. Tenemos que darle un diseño coherente, sencillo, útil y fácil de usar.

El videojuego que pretendemos hacer lo tenemos que diseñar pensando en cómo va a jugar el usuario final. Para ello, primero tendremos que estudiar el perfil del usuario al que va dirigido nuestro producto y las capacidades hardware de la plataforma sobre la que lo desarrollaremos; a partir de este estudio, habrá que definir el diseño de nuestra aplicación. Hay que tener en cuenta que modificar un juego que ya está diseñado para adaptarlo a un tipo de usuario concreto es una tarea muy complicada y debemos evitarla en lo posible.

Para estudiar a nuestro usuario, podemos utilizar el esquema que nos ofrece la Asociación para la Interacción Persona Ordenador (AIPO):

- **Habilidades físicas y sensoriales**. Aunque solemos tener en mente que nuestro mercado será la gente joven (de quince a treinta años), debemos estudiar con detenimiento nuestro público objetivo. Es posible que realicemos videojuegos para preadolescentes, jóvenes que no han jugado previamente a un videojuego o gente adulta. Este estudio determinará aspectos como el tamaño de los botones de la interfaz, los tiempos de espera, el uso del lenguaje...

- **Habilidades cognitivas**. Deberemos diseñar nuestro videojuego pensando en los diferentes usuarios que nos vamos a encontrar: expertos en informática, usuarios tecnológicos, usuarios ocasionales... Habrá que diseñar una interfaz que permita el juego a nuestros usuarios, sea cual sea su capacidad de razonamiento y conocimiento. Es posible que, a partir de esta información, establezcamos diferentes niveles de usuario y presentemos una interfaz acorde con cada uno de ellos.

- **Diferencias de personalidad**. El estudio de la personalidad del grupo de usuarios al que va dirigido nuestro videojuego (timidez, sociabilidad, seguridad...), decidirá el tipo de interrelación que usaremos: acción, tranquilidad, lenguaje coloquial o formal...

- **Diferenciación cultural**. Las expresiones y la terminología que usemos en nuestro videojuego vendrán determinadas por el entorno sociocultural de nuestros usuarios.

Veamos cómo hacer una interfaz gráfica desde la teoría hasta la práctica.

### 2.1.1. Teoría breve del diseño de interfaces gráficas

Para profundizar en el diseño de las interfaces gráficas, debemos conocer los conceptos de metáfora, modelo mental y modelo conceptual. Pasaremos a estudiarlos y después los aplicaremos para realizar un diseño sencillo.

#### Metáfora

La metáfora es la aplicación de una expresión a un objeto o concepto para explicar su semejanza con otro o, según el DRAE:

> "Tropo que consiste en trasladar el sentido recto de las voces a otro figurado, en virtud de una comparación tácita. P. ej. *Las perlas del rocío. La primavera de la vida. Refrenar las pasiones*."

Normalmente, conocemos las metáforas que leemos o escuchamos. Un ejemplo de metáfora puede ser "el mar ha envejecido", donde vemos que se está utilizando un verbo relacionado con los seres vivos (envejecer) sobre un elemento inanimado (mar).

En el diseño de interfaces también se usa el concepto de metáfora para definir comportamientos de elementos que componen nuestra aplicación. La metáfora más conocida es la del escritorio de usuario.

**Escritorio de usuario**

Esta metáfora fue introducida por Xerox y popularizada por Apple.

> En la metáfora del escritorio se define la ventana principal de un ordenador como un escritorio, donde tenemos documentos guardados en carpetas y éstos se pueden editar, copiar, pegar, mover...

Otros ejemplos de metáforas los encontramos en las librerías de controles o, en inglés, *toolkits* (las más populares son Windows Forms, Qt y Gtk). Elementos como un botón, un cuadro de texto, lista de elementos o menú son metáforas clásicas que aparecen en la mayoría de sistemas modernos.

Al interactuar con cualquier elemento de nuestra aplicación, debemos elegir su comportamiento para que sea fácil de usar y requiera un periodo de aprendizaje corto.

En la mayoría de los casos es muy recomendable utilizar metáforas con las que ya se trabaja y que, gracias a su popularidad, permitirán que nuestro usuario sólo necesite adaptar su conocimiento al nuevo uso en nuestra aplicación.

### ¿Dónde utilizar las metáforas?

Las metáforas se utilizan en todos los aspectos de interacción de nuestra aplicación y debemos ser coherentes con su uso. El usuario encontrará las metáforas en los siguientes puntos:

- Elementos para la configuración del juego (número de jugadores, nivel de juego...).
- Elementos del juego: cartas, armas, atrezo...

En una aplicación como puede ser un videojuego, tenemos la licencia de trabajar con interfaces más artísticas y deberíamos aprovecharla.

**Quake de Id Software**

Un ejemplo sería el menú de selección de dificultad que incorporó el videojuego Quake de Id Software. Nada más iniciar el juego, el usuario movía un avatar por un mundo en tres dimensiones. Frente a él tenía tres posibles caminos, donde cada uno simbolizaba un nivel de juego. De esta metáfora podemos destacar:

- Es una forma de selección alternativa que introduce al usuario en el juego desde un principio.
- Es una metáfora elegante.
- El usuario no está acostumbrado a este tipo de interacciones y puede ser reacio y distante en su utilización.

### Ejemplos de metáforas

Un icono es una metáfora muy sencilla que nos representa con un dibujo un objeto que tiene similitudes con el objeto real representado. Los iconos más utilizados son: carpeta, fichero, disquete, CD-ROM, buzón de correo, impresora...

Figura 8. Windows XP Icons © Microsoft

La metáfora más popular es, sin duda, la del escritorio. En la siguiente imagen podemos apreciar un escritorio de un ordenador y nos recuerda a una mesa de trabajo. Contiene iconos de los objetos que solemos tener encima de una mesa.

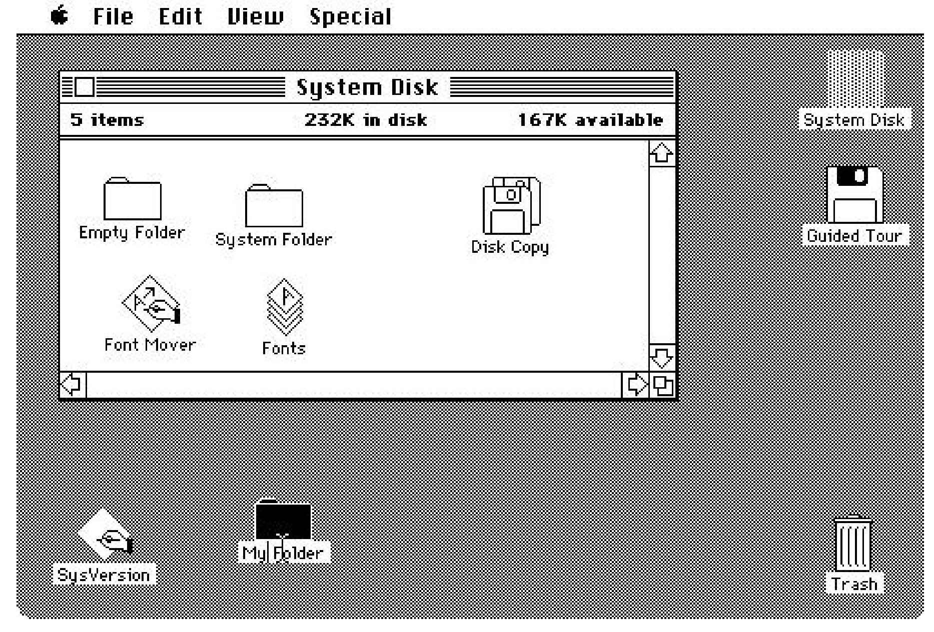



Figura 9. 1984 Mac OS Desktop © Apple

En el caso de los videojuegos, podemos encontrar varias metáforas. Ya de por sí, un escenario 3D es una metáfora visual de un mundo real. Las formas de interactuar con éste también las podemos considerar una metáfora.

Los menús de las aplicaciones también son otra metáfora que representan todas las opciones a nuestra disposición. Se pueden realizar mediante el uso de cadenas de texto o mediante una forma mucho más visual.




Figura 10. Menú de selección de opciones de Half Life © Valve Software

Por otro lado, todo el conjunto de información que se muestra durante el juego también se puede integrar en una metáfora que se conoce por el HUD (Head Up Display). Como veremos posteriormente, el diseño del HUD es clave para proporcionar la mejor experiencia posible al jugador.

En el HUD, la mayoría de juegos se suelen representar por diferentes objetos (enemigos, comida, armas, botiquín de primeros auxilios...) mediante iconos visuales, aunque pueden estar complementados con textos auxiliares. Quizás los HUD más completos y complejos los encontramos en los simuladores, ya que el componente de realismo es muy alto y por tanto hay que proporcionar la máxima cantidad de información al usuario.



Figura 11. Ejemplo del HUD del juego X-Wing Alliance © Lucasarts

### Modelos

Según la psicología, el ser humano almacena el conocimiento en estructuras semánticas. Estas estructuras se organizan a partir de la interacción con el mundo externo usando los procesos perceptuales. Esta información se almacena en diferentes tipos de memoria y se rechaza o se mantiene según diversos criterios, lo que podemos llamar "aprendizaje".

Existen dos modelos que nos permiten estudiar el aprendizaje de un determinado concepto por parte de una persona:

- **Modelo mental**. A partir de las experiencias y el aprendizaje realizado por la persona, éste tiene un modelo inexacto y sólo conoce relaciones sencillas del concepto estudiado ("si se cumple esta condición, ocurre esto"). Por ejemplo, sabe que si presiona el acelerador de un coche, éste se mueve, pero no sabe por qué (no entiende de mecánica).

- **Modelo conceptual**. El estudio del concepto se hace de manera detallada y se conocen todos los mecanismos que actúan sobre él. Es el modelo detallado de un concepto. Siguiendo con el ejemplo anterior, un ingeniero mecánico conoce al detalle cómo está construido un coche.

En el caso de la programación de aplicaciones, el modelo que tiene el programador sobre su aplicación sería un modelo conceptual, mientras que el modelo que se forman los diferentes usuarios sería un modelo mental.

**Web**

Para profundizar más en este tema, os recomendamos visitar la web de la Asociación para la Interacción Persona-Ordenador (www.aipo.es).

Para una misma aplicación, los dos modelos deben asemejarse tanto como sea posible. Cuanta más divergencia haya entre ellos, más probable será el rechazo por parte del usuario. Si queremos que nuestro videojuego sea intuitivo, debemos pensar cómo crear un modelo conceptual para que se asemeje al del usuario.

### 2.1.2. Diseño de interfaces

Los componentes básicos de una interfaz gráfica son:

- Etiqueta
- Cuadro de texto
- Cuadro de confirmación
- Grupo de selección
- Barra de desplazamiento
- Lista de elementos

**Creación de librería**

Aunque nosotros mismos podemos crear nuestra propia librería, sólo daremos una breve explicación de cómo hacerla, ya que este punto queda fuera del alcance del curso.

Podemos utilizar diferentes librerías (*toolkits*) que nos implementan esta serie de componentes, como por ejemplo GTK, CEGUI o QT. En el caso de un videojuego, estos elementos deben de ser muy personalizables para que se integren bien en el diseño global de la aplicación.

#### Etiqueta

Una etiqueta es una representación de un texto sobre nuestra ventana. OpenGL no ofrece ninguna función que permita escribir texto. Por esta razón, algo que aparentemente es tan fácil, debemos implementarlo nosotros.

Las formas más utilizadas para representar texto en OpenGL son:

- Tener el texto renderizado en una imagen y presentarla como textura sobre una superficie. Este punto es muy costoso porque usamos innecesariamente la memoria reservada a las texturas. Por otro lado, es dificultoso hacer una aplicación con varios idiomas usando esta técnica.

- Tener una imagen con el alfabeto (letras, números, signos básicos...) y cargarla como una textura. A partir de aquí renderizaremos varios rectángulos poniendo como texturas la parte de la imagen que corresponda con el signo que se quiere dibujar. Este punto es muy utilizado porque: aprovecha la memoria de las texturas, permite internacionalización, se puede usar la tipografía que se requiera. Por el contrario, la programación puede llegar a ser costosa si la tipografía no es de ancho fijo (por ejemplo, en la tipografía Verdana la 'M' es más ancha que la 'I', pero esto no ocurre en la tipografía Courier).

- Utilizar una librería de *render* de tipografías (TrueType) para generar texturas dinámicas que se utilizarán para presentar el texto. Esta técnica es

correcta y muy útil, pero teniendo en cuenta la vertiente artística de un videojuego, casi siempre es necesario crear tipografías específicas para cada título y se suele optar por la técnica anterior.

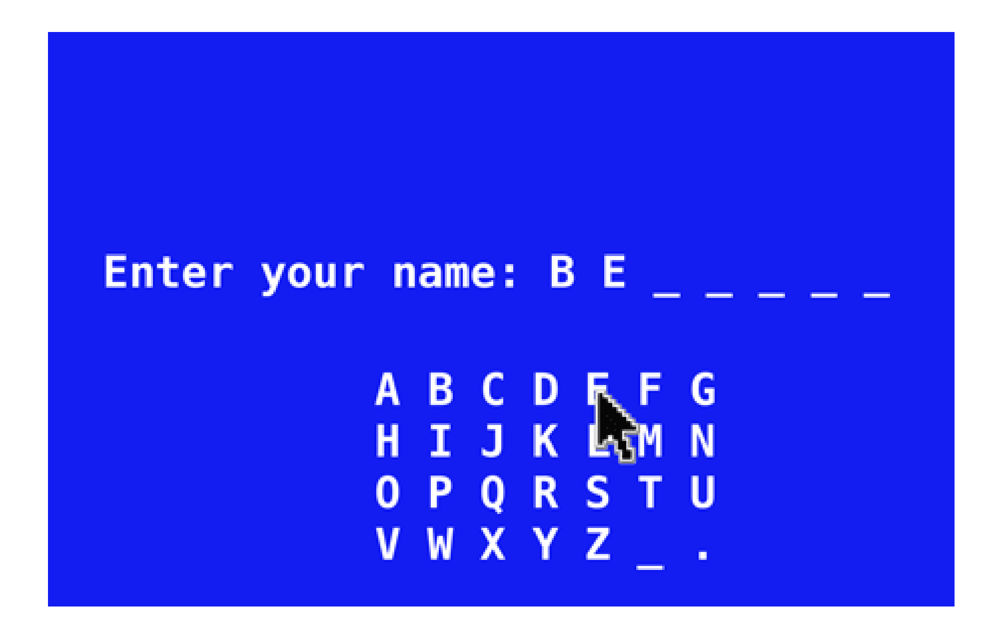
- Un modelo poco usado, pero más elaborado, consiste en crear modelos 3D de cada letra y renderizarlos según consideremos. Este punto es complejo, pero nos permite crear interfaces 3D e incluir efectos a las letras (rotación, escalado, sombras...).

**Cuadro de texto**

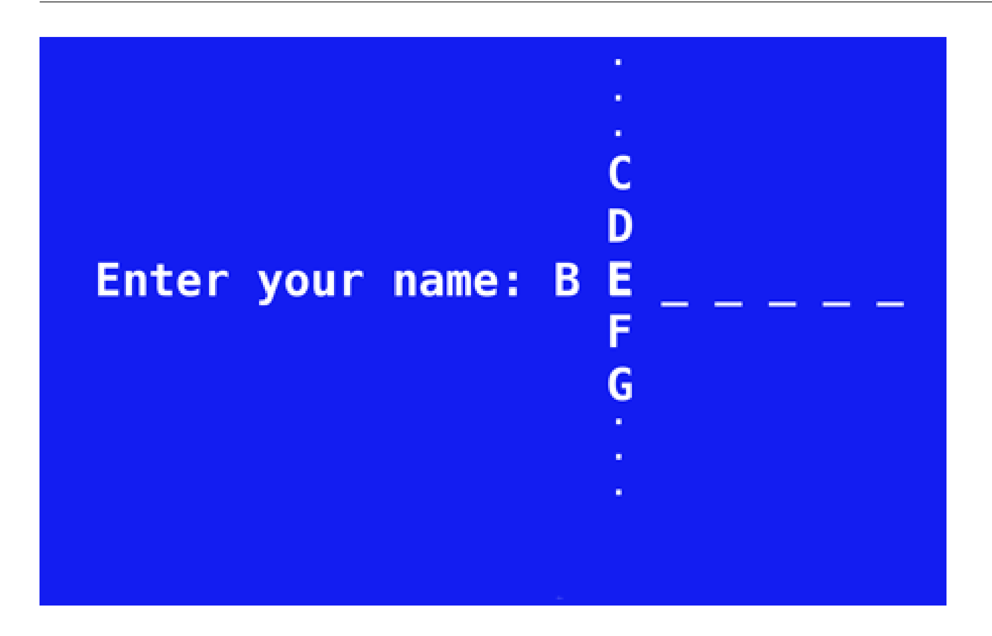
Este elemento gráfico presenta un texto que puede modificar el usuario. En entornos estándares suele utilizarse el teclado para introducir texto, pero cuando se trabaja con videoconsolas, este periférico no suele estar disponible.

Hay dos formas muy utilizadas para introducir texto:

- Se presenta todo el abecedario en la pantalla y, utilizando un apuntador, el usuario va seleccionando letra a letra hasta introducir el texto.




- Se presenta una letra y, a partir de las teclas de cursor, se puede cambiar por la letra correlativa o anterior.



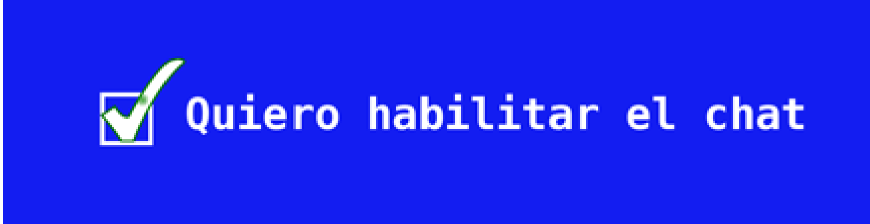

Debemos usar estas dos técnicas dependiendo del contexto y la estética del videojuego. La segunda técnica sería la más correcta si el dispositivo (videoconsola o PC) tiene teclado (ocupa menos espacio en pantalla).

**Cuadro de confirmación**

Estos elementos suelen tener dos posibles estados y suelen simbolizarse con un cuadrado vacío que puede contener un aspa o un indicador para diferenciarlos. En entornos artísticos se tiene la licencia de modificar este comportamiento, pero debe considerarse la simplicidad y representar correctamente la información al usuario.

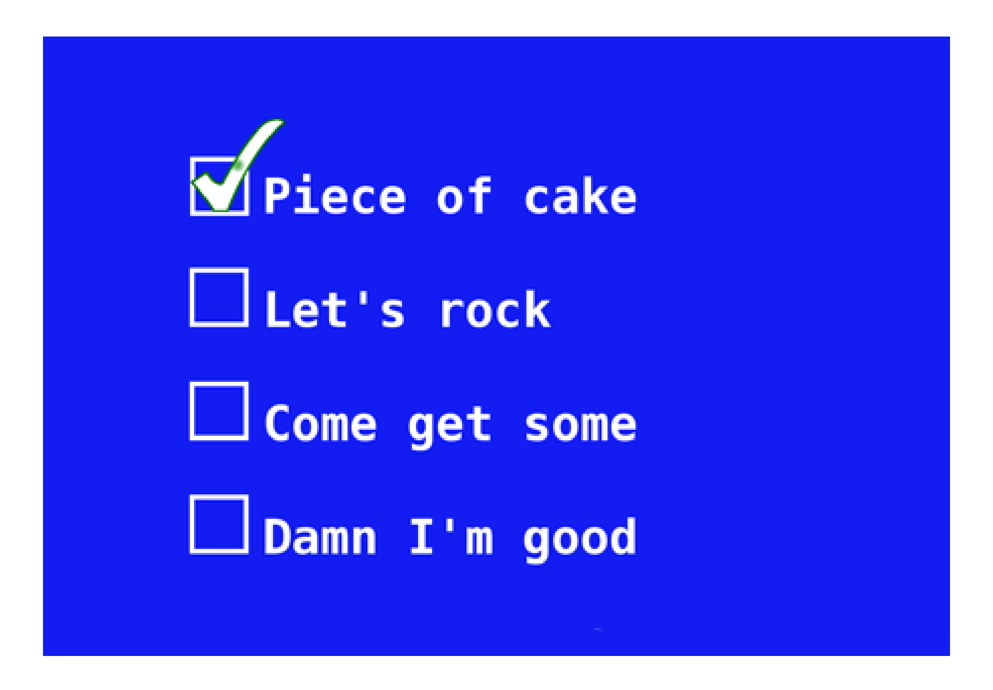
Un error que solemos cometer con este tipo de elementos es la dificultad de entender los dos estados. Por ejemplo, en un cuadro que sólo cambia de color (rojo y verde), no siempre podemos reconocer qué indica cuando está seleccionado.

Normalmente, este elemento está acompañado de una etiqueta que nos indica qué significa esta selección.



**Grupo de selección**

Este elemento es una agrupación de cuadros de chequeo donde se aplica la restricción de que sólo uno puede estar seleccionado.



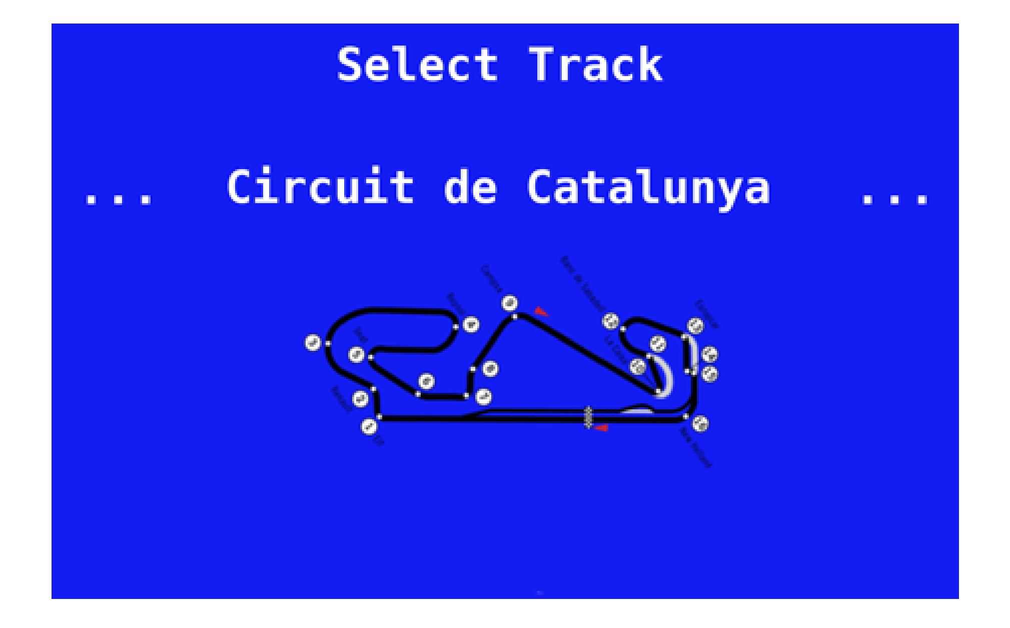

**Lista de elementos**

Este componente presentará al usuario una serie de elementos y éste deberá seleccionar uno (o varios) de ellos. En el caso concreto de un videojuego, esta información se puede presentar de forma más creativa. Algunas posibilidades son:

- Posicionar los elementos (texto, dibujo, animación...) en vertical y resaltar el elemento actual. A partir del cursor, modificar la selección arriba o abajo hasta pulsar con un botón y seleccionar el elemento actual.
  En el caso de que los elementos no quepan en la pantalla, se puede hacer un desplazamiento de los elementos para ir presentándolos todos. Es posible que el elemento que está seleccionado esté siempre centrado en la pantalla.




- Posicionar los elementos en horizontal y trabajar con ellos de manera análoga a la técnica anterior.




En las dos opciones podemos crear una lista circular y volver al elemento inicial una vez lleguemos al último y viceversa.

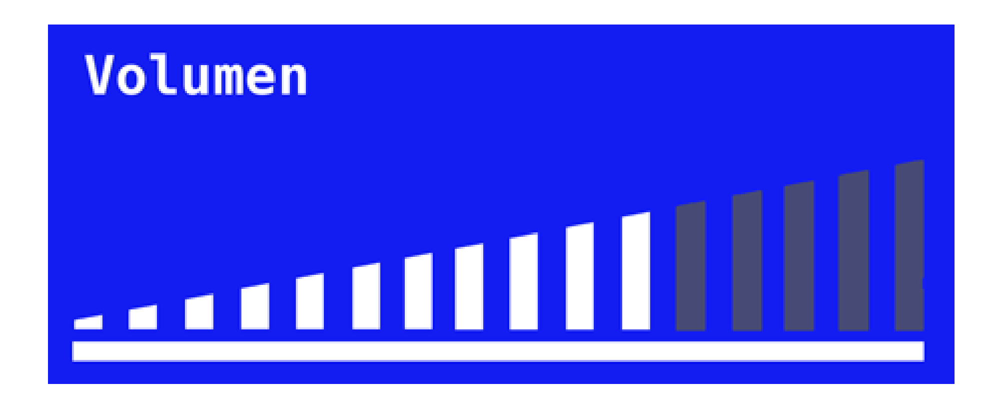
La primera técnica presenta mucha información en la pantalla (sólo vamos a tener en cuenta los idiomas que se escriben en horizontal). Por un lado, permite tener una idea global de lo que se está seleccionando, aunque usualmente no podemos enriquecerlo con información adicional.

En la segunda técnica ocurre lo contrario. Mientras el usuario está seleccionando un elemento, apenas podemos presentar otras opciones, pero tenemos mucho espacio en la pantalla para presentar información adicional al elemento seleccionado (al marcar un vehículo, podemos informar de su velocidad punta, consumo, potencia...).

**Barra de desplazamiento**

Otro elemento que utilizaremos es la barra de desplazamiento. Su metáfora se basa en las barras de desplazamiento que se han utilizado en mecánica y electrónica durante bastantes años (ecualizadores de equipos de música, controladores de volumen, controladores de presión en las máquinas de escribir...).

Consisten en un recorrido normalmente recto o circular que simboliza un número escalar (de cero a cien...). En este recorrido existe un apuntador que puede moverse por el trazado y sirve para seleccionar un valor determinado. Este control suele venir acompañado de una etiqueta de texto que representa el valor seleccionado, como por ejemplo un porcentaje.




### 2.1.3. Diseño de una interfaz para un videojuego

El desarrollo de una interfaz de un videojuego (lo que hemos introducido como el HUD) requiere algunas características especiales que lo diferencian de una interfaz para cualquier otra aplicación. El uso correcto de la interfaz de usuario junto con el diseño de un buen sistema de entrada son claves para facilitar que el usuario aprenda deprisa el funcionamiento del juego y pueda sacar el máximo rendimiento al mismo. Por el contrario, una mala interfaz puede provocar frustración en el jugador, incluso desalentarlo a comprar juegos del mismo desarrollador.

Una medida para conocer el grado de relación existente entre un usuario y la aplicación es lo que conocemos por **usabilidad**. Un sistema con una buena usabilidad nos permitirá:

- Reducir el tiempo que necesitamos para hacer una tarea.
- Reducir el número de errores que cometemos al interactuar con el sistema.

- Reducir el tiempo que necesitamos para aprender a interactuar con el sistema.

También es muy importante que cuando estamos trabajando en el diseño de una interfaz de usuario, nuestra prioridad sea ofrecer toda la información necesaria para el usuario de una forma eficiente, en lugar de centrarnos principalmente en que la información se presente de forma bonita o no. La información que es necesaria mostrar depende del tipo de juego que hagamos (por ejemplo, en un juego de estrategia es abundante, en cambio en una aventura gráfica, es mínima), así que tendremos que estudiar cada caso en particular.

A continuación damos algunas **recomendaciones genéricas** que nos permitirán mejorar la interfaz de usuario de cualquier juego:

- Es importante que sepamos qué decisiones puede tomar el jugador en todo momento, y debemos presentarle en la pantalla toda la información posible para que el usuario pueda decidir cuál es su próximo movimiento. Toda la información relevante se debe mostrar de forma simultánea.

- La información se debe agrupar coherentemente para que el jugador, con una mirada rápida, pueda captar el máximo de información. Una buena estructura en grupos de contenidos nos puede ayudar a presentar mucha más información sin llegar a colapsar al jugador.

- El tiempo de respuesta de la interfaz de usuario tiene que ser mínimo y estar al 100% sincronizado con todos los eventos que suceden en el juego, es decir, no podemos refrescar la interfaz de forma independiente del resto del programa.

- Entre dos diseños de interfaz que nos proporcionen acceso a la misma información, siempre elegiremos el que lo haga de forma más simple e intuitiva.

**¿Textos o botones?**

Al mostrar ciertos elementos en pantalla en los que el usuario puede interactuar, muchas veces nos preguntamos si es mejor mostrar su información en forma de botón o de texto. Vamos a analizar las ventajas y los inconvenientes de ambas opciones:

- Los botones son más fáciles de usar si utilizamos un ratón o un puntero. Los textos acostumbran a ser largos y estrechos, por lo que resultan un objetivo más difícil.

- Por otro lado, los textos tienen un significado claro y directo hacia el usuario y no dejan lugar a interpretación. En cambio, un icono tiene que estar muy bien dibujado y dejar muy claro a qué se refiere para no confundir al usuario, sobre todo cuando se refieren a elementos abstractos.

- En cuanto a la estética, normalmente es más fácil integrar los iconos en el contexto visual del juego.

## 2.2. Sonido

El sonido es un tipo de interacción entre el usuario y el videojuego muy importante, que solemos dejar de lado a favor de los efectos gráficos.

En este apartado estudiaremos qué es el sonido y cómo hace nuestro cuerpo para escucharlo. A partir de la teoría del sonido, veremos las técnicas que tenemos para digitalizar el sonido y poderlo guardar y reproducir en un ordenador. Veremos diferentes tipos de codificación del sonido que nos permitirán ahorrar espacio en disco.

Los últimos puntos los dedicaremos al estudio de una librería de sonido que nos permitirá reproducir:

- Un sonido de fondo
- Efectos especiales
- Sonido en un entorno 3D

### 2.2.1. Teoría del sonido

Vamos a estudiar algunas definiciones de sonido. Según la Wikipedia:

> "El sonido es, desde el punto de vista físico, el movimiento ondulatorio en un medio elástico (normalmente el aire), debido a cambios rápidos de presión, generados por el movimiento vibratorio de un cuerpo sonoro. En general, se llama sonido a la sensación, en el órgano del oído, producida por este movimiento."

Si consultamos el DRAE:

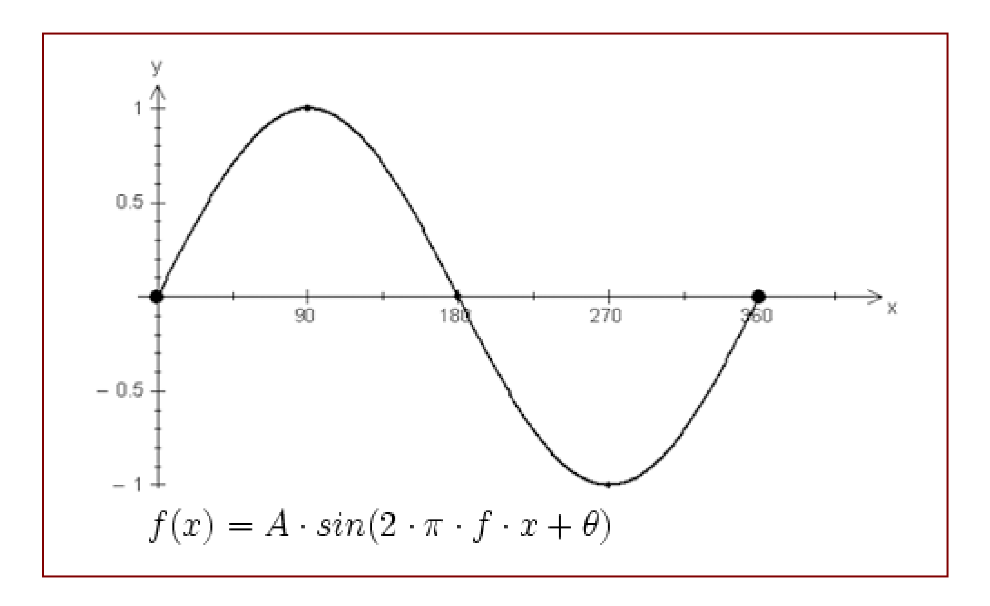
> "Sensación producida en el órgano del oído por el movimiento vibratorio de los cuerpos, transmitido por un medio elástico, como el aire."

#### Características del sonido

El sonido se produce por la vibración de un cuerpo y esta vibración se propaga por el aire que lo rodea. La física nos ha demostrado que estas vibraciones las podemos representar como una suma de infinitas señales senoidales. Para trabajar con el sonido, estudiaremos primero qué información podemos extraer de una señal senoidal:

**Sonido al vacío**

Si tuviésemos las herramientas necesarias, podríamos demostrar la propagación del sonido por el aire introduciendo un reproductor de música en una cápsula, haciendo el vacío dentro y observando que, desde fuera, no se oye nada.




- **Amplitud**. Representada en la ecuación por A. Corresponde a la altura máxima de la senoide. También se suele llamar volumen.

- **Periodo**. Corresponde al tiempo en segundos que tarda la señal en repetirse.

- **Frecuencia**. Es la inversa del periodo y su unidad es el hercio. La frecuencia es el número de ciclos por segundo de la señal. En la ecuación anterior se representa por la letra *f*.

- **Longitud de onda**. Es el tamaño que mide la señal cuando se transmite por el medio.

- **Fase**. Representa la posición relativa entre dos ondas. Su unidad son los radianes (o grados). En la ecuación anterior se representa por la letra $\theta$.

Los músicos utilizan otras características para trabajar con el sonido:

- **Altura**. Está relacionado con la frecuencia del sonido. Es un parámetro que se utiliza para determinar el tono de un sonido. La altura puede ser alta o baja.

- **Tono**. Se define como agudo o grave según la frecuencia del sonido.

- **Nota musical**. Se definen siete notas básicas que están relacionadas con una frecuencia de sonido concreto. Tanto esta característica como las dos anteriores son diferentes representaciones de la frecuencia.

- **Duración**. Es el tiempo en que un instrumento mantiene una nota.

- **Intensidad**. Representa la amplitud de la señal. Es sinónimo de volumen.

- **Timbre**. Es la cualidad de un instrumento que permite distinguir su sonido del de otro.

**Aplicaciones al mundo real**

A partir de las características anteriores, estudiaremos diversas aplicaciones donde podemos aprovecharlas.

El caso más sencillo es subir o bajar el volumen a un sonido. Si somos capaces de modificar la amplitud de una onda, podremos cambiar su volumen. Matemáticamente se puede expresar como la multiplicación de la onda por un factor:

$$f'(x) = factor \cdot f(x) = factor \cdot A \cdot sin(2 \cdot \pi \cdot f \cdot x + \theta)$$

Modificando la frecuencia de una señal, podemos hacer que un sonido sea más agudo o más grave. La teoría matemática que hay detrás es más compleja que la anterior, pero, como veremos en el siguiente punto, una vez trabajemos con una representación digital del sonido, será sencillo modificar su frecuencia.

Cuando una fuente de sonido está en movimiento, la frecuencia del sonido que oye el receptor depende de su posición. Si la fuente se mueve en dirección al receptor, éste oirá el sonido con una frecuencia más alta que el sonido original. Si la dirección fuese opuesta, el sonido tendría una frecuencia más baja.

Lo podemos observar al oír el sonido de un tren cuando viene hacia nosotros y después se aleja.

Este efecto se conoce como efecto Doppler, enunciado por Christian Andreas Doppler en 1842.

Otra aplicación, en este caso de ciencia ficción, trata sobre la fase. Sumar dos senoides de fases opuestas no genera sonido:

$$A \cdot sin(2 \cdot \pi \cdot f \cdot x) + A \cdot sin(2 \cdot \pi \cdot f \cdot x + \pi) = 0$$

En este caso, si pudiésemos generar una señal idéntica pero con una fase opuesta, podríamos dejar de oír el sonido que quisiésemos, como por ejemplo el ruido molesto de unas obras, una carretera o un aeropuerto. Aunque, para poder generar una señal idéntica, necesitaríamos conocer el futuro.

La transformada de Fourier nos proporciona un algoritmo que nos permite obtener los coeficientes de las diferentes funciones sinodales que componen una señal compleja. El algoritmo que podemos utilizar para obtener los coeficientes es el FFT (*fast Fourier transform*), aunque su explicación se aleja del objetivo de este módulo.

A partir de las diferentes amplitudes obtenidas con la transformada de Fourier, podemos conocer el espectro del sonido y, al modificar estos valores y volver a reconstruir la señal, crear un ecualizador.

### 2.2.2. Codificación del sonido

Para poder trabajar con el sonido, tenemos que poder representarlo digitalmente. Existen las siguientes maneras de guardar sonido:

- Guardar secuencias de notas de diferentes instrumentos teóricos. Se guarda una relación de todas las notas que componen la sintonía (duración, frecuencia, amplitud).

- Utilizar un ADC (*analogic-digital converter*) para muestrear un determinado sonido y guardarlo como una secuencia de bytes que aproximan la onda original.

- A partir de sonido digitalizado, crear pequeñas muestras de sonido de instrumentos y crear partituras que los utilicen. Este punto sería un híbrido de los dos puntos anteriores.

#### Formato MIDI

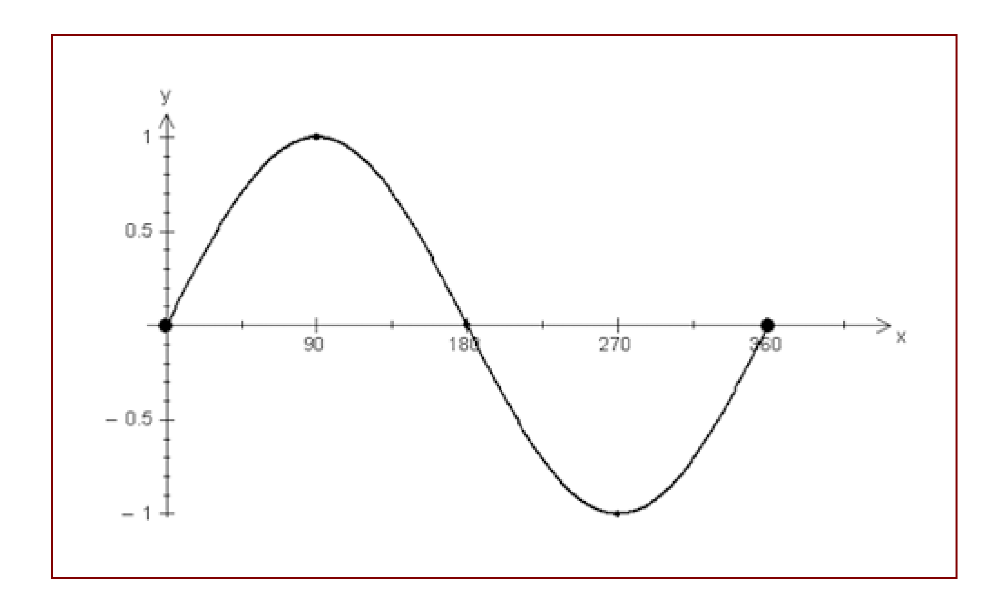
El formato MIDI mantiene una estructura como la citada en el primer tipo. Se publicó su primera especificación en 1983 y establece un protocolo donde se definen variables que determinan cómo deben sonar las notas de diversos instrumentos. Para poder utilizar este formato, necesitamos crear el sonido de los diferentes instrumentos por software. Esta forma de trabajo se popularizó con los sintetizadores digitales, que recreaban el sonido de varios instrumentos utilizando circuitos electrónicos.

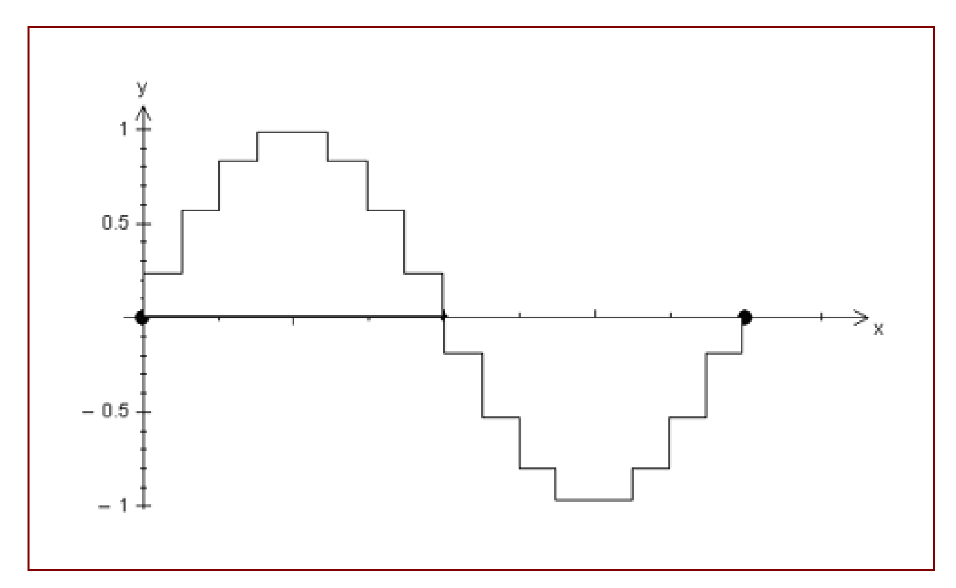
Este tipo de formato ocupa poco espacio de memoria, pero necesita de un algoritmo que genere el sonido de los diferentes instrumentos. No obstante, la mayoría de las tarjetas de sonido incluyen un chip que hace este trabajo por nosotros.

## Digitalización

En electrónica es muy común utilizar dispositivos ADC (*analogic-digital converters*) o conversores de analógico a digital. Esta electrónica transcribe la información analógica en información digital. En este proceso es inevitable perder información, pero, dependiendo del algoritmo y diferentes parámetros, esta pérdida puede ser despreciable.

El algoritmo más común es el llamado **PCM**, que consiste en realizar una muestra cada cierto tiempo y codificar la lectura en una serie de bits.




**Ejemplo**

La calidad de un CD de audio es de 44.100 muestras por segundo (o 44.100 Hz), la muestra se codifica en 16 bits y la conversión se hace sobre dos canales (estéreo).

El **formato WAV** permite guardar una pequeña cabecera, donde se informa de la frecuencia que se ha utilizado para la digitalización, el tamaño en bits de la muestra y el número de canales. Después de la cabecera, se almacena la información digital que representa el sonido sin ningún tipo de compresión. Este tipo de ficheros ocupan mucho espacio de memoria, pero reproducirlos es tan sencillo como enviar la información directa del fichero a un dispositivo DAC (*digital-analogic converter*). Las tarjetas de sonido son dispositivos que contienen tanto un ADC como un DAC incorporados.

**Espacio de memoria**

Un minuto con calidad de CD ocupa 10 MB.

Para evitar un uso tan exagerado de disco (y red en el caso de enviar audio por Internet), se popularizó el **formato MP3** a principios de los años noventa. Este formato comprime con pérdidas la información del sonido. El algoritmo permite hacer un balanceo entre la calidad y el tamaño de compresión. A mayor compresión, mayor pérdida. Un sonido con una calidad aceptable puede ocupar hasta diez veces menos que su original.

Se han creado los formatos **Ogg Vorbis** y **Windows Media Audio**, que quieren competir con el MP3. Para poder reproducir sonido que está guardado en estos formatos, necesitaremos librerías que descompriman la información y enviar el sonido reconstruido a un DAC. Este tipo de formatos nos permiten aprovechar mejor el espacio en disco, pero nos añaden un coste en tiempo de cálculo para poder descomprimir el sonido.

### Trackers

En la década de los ochenta se popularizaron los audios generados con secuenciadores (o *trackers*).

> Un *tracker* es una aplicación donde introducimos una serie de muestras (*samples*) o sonidos cortos que trataremos como instrumentos (como ejemplos típicos tenemos el bombo y la caja de una batería, una trompeta, cuerdas de bajo o guitarra).

El programa proporciona una serie de patrones (*patterns*) con los que podemos hacer que un instrumento suene en un determinado tiempo con una determinada nota de forma parecida a lo que se hace con el formato MIDI.

Este tipo de formato fue muy importante cuando el espacio de disco era muy limitado, ya que por un lado lo aprovechaban al máximo como el formato MIDI y, por el otro, si el compositor trabajó duro, podía tener una calidad parecida a un sonido digitalizado. Como anécdota, este tipo de formatos permitieron crear música electrónica con relativa facilidad.

**Formatos de *trackers***

Como formatos concretos tenemos MOD y XM.

Este tipo de aplicaciones se siguen utilizando para componer música, aunque cada vez es más normal generar, una vez se haya finalizado la composición, un fichero MP3 para reproducirlo en un videojuego.

### 2.2.3. Historia del sonido en los videojuegos

Los primeros videojuegos de la historia tenían una capacidad sonora muy limitada. Por ejemplo, el Atari 2600 sólo era capaz de generar dos notas o tonos simultáneamente, mientras que los primeros ordenadores personales disponían de un único altavoz interno de 1 bit que sólo generaba los famosos sonidos conocidos como *beeps*.

**Música en los primeros videojuegos**

En los primeros videojuegos la música se encontraba almacenada en secuencias de datos con las notas. A veces los programadores tenían que introducir "a mano" nota por nota para poder crear una melodía. A principios de los ochenta se creó la figura del compositor musical para videojuegos, junto con melodías tan famosas como la de Super Mario Bros, obra de Koji Kondo.

El principal avance en la evolución del sonido fue la introducción de la tarjeta de sonido como un elemento de proceso independiente. Las primeras videoconsolas y algunos ordenadores personales (Commodore Amiga o Atari ST) incorporaban tarjetas que permitían mezclar la onda de varios instrumentos para crear melodías más complejas.

Una tarjeta de sonido es, básicamente, un conversor de digital-analógico, llamado DAC por sus siglas en inglés. Este dispositivo traduce una serie de valores digitales (bits) en tensiones eléctricas. Solemos conectar estas tensiones eléctricas a un electroimán que hará vibrar una membrana y generará sonidos (altavoz).

Podemos crear una tarjeta de sonido rudimentaria conectando unas cuantas resistencias a un puerto paralelo del ordenador e ir escribiendo en él, con una frecuencia determinada, diferentes valores.

Las primeras tarjetas de sonido comerciales incluían un conversor DAC en una tarjeta interna y los valores se enviaban usando un puerto de entrada/salida. Más tarde, los fabricantes aprovecharon los DMA de los procesadores para transmitir esta información. La evolución pasó por la resolución del DAC (de 8 bits a 16 y 32) e incluir nuevos canales (mono, estéreo, 4.1, 5.1).

Quizás el avance más importante vino de la mano de empresas como Adlib o Creative Labs, que presentaron tarjetas que se podían comprar de forma independiente e instalar en cualquier ordenador personal.

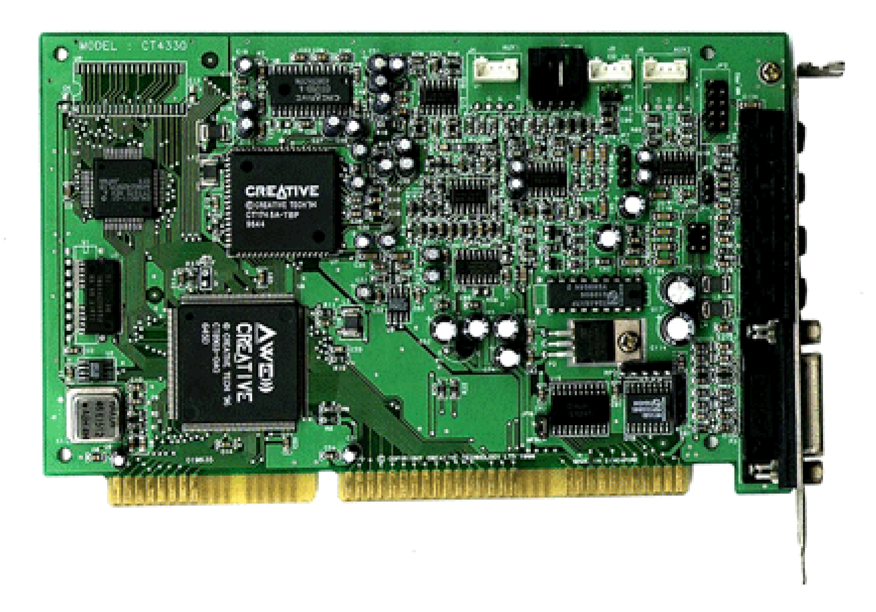



Figura 12. Creative Sound Blaster AWE 32 © Creative Labs

La evolución de las tarjetas de sonido fue permitiendo añadir más canales y mejor calidad, mejorando la sensación de inmersión del usuario. Pero el salto cualitativo más importante de los últimos años se realizó de la introducción de sistemas de decodificación digital, como Dolby Surround, en el mundo de los videojuegos. Este tipo de tecnologías nos permiten codificar información espacial y reproducirla en varios altavoces (normalmente entre 4 y 7), dando una sensación de realismo

El tratamiento del sonido en las consolas de última generación ha supuesto uno de los avances más importantes en cuanto a la calidad de los juegos. Cada uno de los fabricantes de consolas de última generación ha elegido su propio sistema de codificación, como podemos observar a partir de las especificaciones publicadas para cada una de ellas:

- La videoconsola Xbox 360 permite trabajar con muestras de 16 bits a 48kHz, hasta 256 canales simultáneos y soporta de inicio descompresión del sistema Dolby Digital.
- La videoconsola Playstation 3 soporta Dolby Digital 5.1 surround.
- La videoconsola Nintendo Wii incluye soporte para Dolby Pro Logic II.

### 2.2.4. Introducción a las librerías de sonido

Paralelamente a la evolución de los dispositivos de audio, también han ido apareciendo y mejorando librerías para trabajar con sonido mediante este hardware específico. Podríamos hablar de los siguientes **tipos de librerías**:

- **Bajo nivel**. Las que permiten trabajar directamente con el DAC (enviar datos para reproducir sonidos). No encontraremos mucha oferta, ya que la mayoría se crean y se utilizan por el mismo programador.

- **Medio nivel**. Las que permiten reproducir más de un sonido mezclándonos según nuestro interés. Tenemos librerías sencillas que pueden sernos útiles para determinados tipos de juegos. La librería SDL contiene una API de sonido que podemos incluir en este tipo.

- **Alto nivel**. Las que permiten reproducir sonido en tres dimensiones y generar efectos sobre el canal de salida. Incluimos la librería DirectSound de Microsoft y OpenAL. Este tipo de librerías nos presenta una API con la que podemos reproducir sonidos, mezclarlos y posicionarlos en un espacio 3D. Proporciona funciones para cambiar atributos del sonido (volumen, frecuencia...) y crear efectos (efecto *doppler*, reverberación...).

La librería EAX de CreativeLabs nos proporciona las herramientas para modificar el sonido que generamos según el entorno donde se produzca. Por ejemplo, el hardware nos permite simular el sonido como si estuviésemos en una cueva, en un auditorio o al aire libre.

En el mercado encontramos también otro tipo de librerías que nos ofrecen una capa de abstracción para poder trabajar con diferentes librerías dependiendo de la plataforma de desarrollo. Un claro ejemplo es **FMOD**, una librería de sonido que nos ofrece todas las funcionalidades descritas anteriormente para las plataformas: XBox, XBox360, PlayStation 2, PlayStation Portable, PlayStation 3, GameCube, Wii y PC (Windows, Mac y Linux).

La principal ventaja que tenemos al utilizar una librería como FMOD es que nos proporciona una API sencilla para el desarrollo en cualquiera de las plataformas que soporta. También ofrece la garantía de no tener fallos de programación porque es un código muy probado.

Por el contrario, debemos adquirir una licencia de la librería si queremos comercializar nuestra aplicación, aunque si contásemos los costes que nos supondría crear una librería de sonido, portarla y probarla en diversas plataformas, probablemente no nos saldría a cuenta.

Los pasos correctos para usar esta librería son:

- Configurar nuestro compilador para que pueda incluir los ficheros de cabecera.
- Configurar nuestro compilador para que pueda enlazar nuestro código con el fichero objeto de la librería.
- Incluir el código de inicialización de la librería en nuestro programa.

- Incluir el código de carga de ficheros de sonido y su reproducción.

En el caso concreto de la librería FMOD, el código de inicialización sería:

```
fmodtest.cpp
#include <fmod.hpp>
#include <fmod_errors.h>
#include <iostream>
#include <windows.h>

static void checkError(FMOD_RESULT result)
{
   if (FMOD_OK != result)
   {
      std::cout << FMOD_ErrorString(result) << std::endl;
      exit(-1);
   }
}

void initFMOD()
{
   FMOD::System* system;
   FMOD_RESULT result;
   unsigned int version;
   FMOD_CAPS caps;
   FMOD_SPEAKERMODE speakerMode;

   checkError(FMOD::System_Create(&system));
   checkError(system->getVersion(&version));
   if (version < FMOD_VERSION)
   {
      std::cout << "Version erronea" << std::endl;
      exit(-1);
   }
   checkError(system->getDriverCaps(0, &caps, 0, &speakerMode));
   if (caps & FMOD_CAPS_HARDWARE_EMULATED)
      checkError(system->setDSPBufferSize(1024, 10));
   result = system->init(100, FMOD_INIT_NORMAL, 0);
   if (result == FMOD_ERR_OUTPUT_CREATEBUFFER)
   {
      checkError(system->setSpeakerMode(FMOD_SPEAKERMODE_STEREO));
      checkError(system->init(100, FMOD_INIT_NORMAL, 0));
   }
}
```

### 2.2.5. Reproducción de música

Siempre es importante crear un ambiente en un videojuego y para ello es muy recomendable reproducir música de fondo. Las librerías de sonido nos permiten cargar un fichero de sonido en memoria. La codificación del fichero puede ser de lo más variada y siempre podemos usar programas externos para convertir nuestra música a cualquier formato.

A partir de esta zona de memoria, podemos pedirle a la librería que reproduzca el sonido, al que podemos:

- Subir y bajar el volumen.
- Cambiar la frecuencia de muestreo (más agudo o más grave).
- Escuchar en bucle (a partir de un tiempo determinado, volver a empezar).

Podemos ir cambiando la música a medida que ocurran eventos (zona de peligro, aire libre, acción...) y para ello vamos a estudiar cómo hacer este tipo de transacciones.

Lo que debemos tener claro es que no podemos cambiar de un sonido a otro sin hacer ningún tipo de transición. Este cambio es bastante molesto, llama demasiado la atención del jugador y deja de estar inmerso en la trama.

Debemos crear transiciones entre sonidos. El más común es entremezclar los dos sonidos e ir bajando el volumen al sonido que queremos quitar (*fade-out*) e ir subiendo progresivamente el volumen al nuevo sonido (*fade-in*). Para ello necesitamos, por un lado, que la librería permita mezclar sonidos (*mixer*) y, por el otro, poder subir y bajar el volumen a los sonidos.

**Control del tiempo**

Tenemos que controlar correctamente el tiempo en nuestra aplicación para que la música suene perfectamente y todas las transiciones sean correctas.

Vamos a estudiar cómo se realizan todas estas tareas mediante la librería FMOD. En primer lugar necesitamos conocer cómo cargar los sonidos en la memoria:

```
FMOD::Sound* sound;
FMOD::Channel* channel;
checkError(system->createSound("sonido.mp3", FMOD_3D, 0, &sound));
checkError(system->playSound(FMOD_CHANNEL_FREE, sound, true, &channel));
```

Una vez tenemos en la memoria el sonido que queremos reproducir, podemos iniciar o parar la reproducción en cualquier momento mediante la siguiente función:

```
checkError(channel->setPaused(false));
```

Para subir y bajar el volumen:

```
checkError(channel->setVolume(1.0f));
```

### 2.2.6. Reproducción de efectos especiales

Una vez tenemos la música de fondo, habrá que llevar el control de los diferentes efectos especiales que se irán oyendo en el juego, por ejemplo, disparos, rayos, explosiones, puertas, coches, pelotas...

La idea más sencilla es tener un conjunto de los sonidos que se están reproduciendo. A medida que van finalizando, los vamos sacando del conjunto y, cuando ocurre un evento que necesita un sonido, crearlo e introducirlo en el conjunto.

Al cargar los sonidos, debemos tener en cuenta que el acceso a disco puede llegar a ser costoso, así que lo mejor es tener los sonidos que vamos a utilizar cargados en memoria, pero teniendo en cuenta que ocuparán bastante espacio.

Como en la reproducción de música, es primordial tener un control absoluto del tiempo para que el sonido se reproduzca perfectamente y no haya problemas con el inicio y fin de cada uno de los efectos.

Debemos tener acceso a estos sonidos y poder modificar sus parámetros tratados en el anterior apartado. Por ejemplo, el sonido de un coche siempre va a estar oyéndose (crearlo en bucle) y deberemos subir o bajar el volumen según la distancia y subir o bajar la frecuencia según el número de revoluciones de su motor (o velocidad).

En la siguiente clase "UOCSoundManager", presentamos una base para crear un controlador como el que se ha explicado en este apartado:

**UOCSoundManager.h**

```
#ifndef __UOCSoundManager__
#defone __UOCSoundManager__

#include <fmod.hpp>
#include <fmod_errors.h>
#include <map>
#include <string>

class UOCSoundManager
{
public:
   UOCSoundManager();
   ~UOCSoundManager();
```

```
public:
   void init();
   void done();

   void update();
   void createSound(const std::string& name, const std::string& filename);
   int playSound(const std::string& name);
   FMOD::Channel* getChannel(int playId);

private:
   static void checkError(FMOD_RESULT result);

private:
   FMOD::System* m_system;
   std::map<std::string, FMOD::Sound*> m_sounds;
   std::map<int, FMOD::Channel*> m_playingSounds;
};
#endif // __UOCSoundManager__
```

### UOCSoundManager.cpp

```
#include "UOCSoundManager.h"
#include <cstdlib>
#include <iostream>

UOCSoundManager::UOCSoundManager()
{
}

UOCSoundManager::~UOCSoundManager()
{
}

void UOCSoundManager::init()
{
   FMOD_RESULT result;
   unsigned int version;
   FMOD_CAPS caps;
   FMOD_SPEAKERMODE speakerMode;

   checkError(FMOD::System_Create(&m_system));
   checkError(m_system->getVersion(&version));
   if (version < FMOD_VERSION)
   {
      std::cout << "Version erronea" << std::endl;
      exit(-1);
```

```
   }
   checkError(m_system->getDriverCaps(0, &caps, 0, 0, &speakerMode));
   if (caps & FMOD_CAPS_HARDWARE_EMULATED)
      checkError(m_system->setDSPBufferSize(1024, 10));
   result = m_system->init(100, FMOD_INIT_NORMAL, 0);
   if (result == FMOD_ERR_OUTPUT_CREATEBUFFER)
   {
      checkError(m_system->setSpeakerMode(FMOD_SPEAKERMODE_STEREO));
      checkError(m_system->init(100, FMOD_INIT_NORMAL, 0));
   }
}

void UOCSoundManager::done()
{
}

void UOCSoundManager::update()
{
   m_system->update();
}
```

En el siguiente método sólo se crea la estructura necesaria para guardar un sonido en memoria. En ningún momento se está reproduciendo. La clase que estamos desarrollando guardará una tabla relacionando un nombre que le indicamos por parámetro con la estructura de FMOD que representa el sonido.

```
void UOCSoundManager::createSound(const std::string& name, const std::string& filename)
{
   FMOD::Sound* sound;
   checkError(m_system->createSound(filename.c_str(), FMOD_3D, 0, &sound));
   m_sounds[name] = sound;
}
```

En el método "playSound" obtendremos la estructura de FMOD que representa el sonido, y que hemos guardado previamente en la tabla "m_sounds", y lo reproduciremos en un canal libre de FMOD:

```
int UOCSoundManager::playSound(const std::string& name)
{
   FMOD::Sound* sound = m_sounds[name];
   FMOD::Channel* channel;
   checkError(m_system->playSound(FMOD_CHANNEL_FREE, sound, true, &channel));
   static int nextID = 0;
   int id = nextID++;
   m_playingSounds[id] = channel;
   return id;
```

```
}
```

Para poder trabajar con un canal (cambiar volumen, cambiar frecuencia...), necesitaremos una referencia a la estructura que mantiene FMOD:

```
FMOD::Channel *UOCSoundManager::getChannel(int playId)
{
   std::map<int, FMOD::Channel*>::iterator found = m_playingSounds.find(playId);
   if (found == m_playingSounds.end())
         return NULL;
   return found->second;
}

void UOCSoundManager::checkError(FMOD_RESULT result)
{
   if (FMOD_OK != result)
   {
         std::cout << FMOD_ErrorString(result) << std::endl;
         exit(-1);
   }
}
```

Un ejemplo de uso de este controlador puede ser el siguiente. Tenemos un controlador de dispositivos de entrada que nos indicará cuándo una tecla está pulsada. Aprovecharemos esta información para aumentar o disminuir la frecuencia de un sonido de coche:

**Ejemplo de un bucle de juego**

```
...
void GameEngine::init()
{
   ...
   m_inputManager.init(hWnd, WINDOW_WIDTH, WINDOW_HEIGHT);
   m_soundManager.init();
   m_soundManager.createSound("coche", "d00222.mp3");
   m_cocheId = m_soundManager.playSound("coche");
   m_soundManager.getChannel(cocheId)->setPaused(false);
}
```

El siguiente método se utilizará en cada bucle del juego, y veremos si se han pulsado las teclas del cursor arriba y abajo. Según estas pulsaciones, subiremos o bajaremos la frecuencia del canal que está reproduciendo el sonido de un coche. Remarcamos que la frecuencia subirá o bajará según el valor "timeSinceLastFrame" que representa, en este caso, el número de milisegundos que

han transcurrido desde la anterior llamada. De esta manera independizamos el comportamiento de nuestro código de los demás procesos y de la velocidad del hardware.

```
void GameEngine::onFrame(float timeSinceLastFrame)
{
   m_inputManager.capture();
   m_soundManager.update();
   if (m_inputManager.getKeyboard().isKeyDown(OIS::KC_UP))
   {
      FMOD::Channel* channel = m_soundManager.getChannel(m_cocheId);
      float freq;
      channel->getFrequency(&freq);
      std::cout << "Freq: " << freq << std::endl;
      freq += timeSinceLastFrame;
      if (freq > 60000.0f)
         freq = 60000.0f;
      channel->setFrequency(freq);
   }
   if (inputManager.getKeyboard().isKeyDown(OIS::KC_DOWN))
   {
      FMOD::Channel* channel = m_soundManager.getChannel(m_cocheId);
      float freq;
      channel->getFrequency(&freq);
      std::cout << "Freq: " << freq << std::endl;
      freq -= timeSinceLastFrame;
      if (freq < 30000.0f)
         freq = 30000.0f;
      channel->setFrequency(freq);
   }
}
```

### 2.2.7. Posicionamiento tridimensional del sonido

En un videojuego con capacidades gráficas tridimensionales, es necesario que el sonido se reproduzca según la posición en el mundo virtual del emisor. Aprovechando el sonido estéreo o multicanal de las tarjetas actuales, podemos reproducir el sonido haciendo creer al oyente que el emisor está en una determinada posición (por ejemplo, subir el volumen en el canal izquierdo y bajar el derecho para hacer creer que está a la izquierda). También podemos jugar con el volumen para simular la distancia entre el emisor y el receptor.

Para poder crear un motor sonoro, es necesario conocer las propiedades físicas del sonido y cómo lo escucha el ser humano. Por suerte, la mayoría de librerías actuales nos ofrecen una infraestructura que permite crear sonido en tres dimensiones con muy buena calidad.

Para reproducir un sonido en tres dimensiones, necesitamos:

- Posicionar y orientar el receptor (*listener*) en el espacio.
- Posicionar y orientar la fuente del sonido (emisor) usando la misma base de coordenadas.
- Definir el sonido y volumen.

Las librerías también nos permiten informar a los emisores y receptores de su velocidad para que pueda simular el efecto *doppler*.

Aprovechando el ejemplo del apartado anterior, situaremos el sonido del coche en la misma posición que está el ratón y al oyente, en el centro de la ventana, así notaremos el efecto del sonido 3D:

**Modificación a UOCSoundManager.h**

```
...
   FMOD::System* getSystem() { return m_system; }
...
```

**Ejemplo de un bucle de juego**

```
...
void GameEngine::init()
{
   ...
   m_inputManager.init(hWnd, WINDOW_WIDTH, WINDOW_HEIGHT);
   m_soundManager.init();
```

En las siguientes líneas situamos al oyente dentro de un mundo virtual. Aunque lo normal es relacionar estas coordenadas con el motor gráfico (y físico), las librerías que intervienen no comparten esta información. Como en el caso de las cámaras de la librería gráfica, el oyente se define por una posición y dos vectores más (vector "arriba" y hacia donde se mira) que definen correctamente la orientación. También se informa de su velocidad para poder simular el efecto *doopler*:

```
   FMOD_VECTOR pos = { 0.0f, 0.0f, 0.0f };
   FMOD_VECTOR vel = { 0.0f, 0.0f, 0.0f };
   FMOD_VECTOR forward = { 0.0f, 0.0f, 1.0f };
   FMOD_VECTOR up = { 0.0f, 1.0f, 0.0f };
   m_soundManager.getSystem()->set3DListenerAttributes(0, &pos, &vel, &forward, &up);

   m_soundManager.createSound("coche", "coche.mp3");
   m_cocheId = m_soundManager.playSound("coche");
   m_soundManager.getChannel(cocheId)->setPaused(false);
```

```
}
```

En el caso de este ejemplo, leemos la posición del ratón, tratamos un poco la posición para cambiar de coordenadas de pantalla a coordenadas del mundo virtual, y modificamos la posición (y velocidad) de la fuente del sonido:

```
void GameEngine::onFrame(float timeSinceLastFrame)
{
   m_inputManager.capture();
   m_soundManager.update();
   float x = m_inputManager.getMouse().getMouseState().X.abs - WINDOW_WIDTH / 2.0f;
   float z = m_inputManager.getMouse().getMouseState().Y.abs - WINDOW_HEIGHT / 2.0f;
   FMOD_VECTOR pos = { x, 0.0f, z };
   FMOD_VECTOR vel = { 0.0f, 0.0f, 0.0f };
   m_soundManager.getChannel(m_cocheId)->set3Dattributes(&pos, &vel);
}
```

## 2.3. *Feedbacks*

Los fabricantes de dispositivos de entrada han ido introduciendo mejoras en sus productos para crear nuevas sensaciones en el jugador. En este apartado nos dedicaremos al estudio de los dispositivos de respuesta al usuario, o *feedback*.

El dispositivo de *feedback* más implantado es el de fuerza. Permite que el videojuego recree sensaciones de fuerza sobre el *joystick*. Con este dispositivo podemos:

- Recrear un temblor, por ejemplo, para simular una salida de carretera en un juego de carreras.
- Hacer fuerza en una dirección determinada, por ejemplo, cuando en un juego de simulación de vuelo la avioneta está ascendiendo o girando.
- Decidir la cantidad de fuerza que se requiere en cada una de las acciones anteriores.
- En el caso de la vibración, podemos indicar el periodo o frecuencia de la misma; por ejemplo, aumentar la frecuencia de vibración a medida que el terreno se haga más rugoso.

No hay muchas librerías que nos permitan programar el comportamiento de estos dispositivos. La más conocida es DirectInput, que nos proporciona una API sencilla para generar estas respuestas, aunque también existen implementaciones no oficiales para SDL.

**Mandos de consola**

En los mandos de consola también suele incorporarse un *feedback* de vibración, aunque su integración en las consolas de última generación ha sido especialmente complicada, ya que se ha mezclado la vibración con sensores de movimiento.

# 3. Videojuego a videojuego

El último tipo de comunicación que vamos a estudiar es el que se produce entre diferentes ordenadores o videoconsolas a través de una red. Usando esta conexión, podemos interactuar con otros jugadores y participar de un modo cooperativo o competitivo en un videojuego.

La inclusión de tecnologías de red en un juego se ha convertido en un elemento clave para incrementar las ventas y la calidad del mismo. El hecho de añadir una opción multijugador aumenta su tiempo de vida, es decir, el tiempo que el jugador puede utilizar el juego (sin aburrirse), y por tanto aumenta el valor que el usuario percibe de un juego.

El potencial ofrecido por Internet como red de comunicación global ha revolucionado en gran parte el uso de las tecnologías de red en los videojuegos, principalmente porque proporciona acceso a un 70% de la población de los países más desarrollados. Empezaremos por analizar resumidamente la tecnología que hay tras Internet para saber cómo se realiza la transferencia de información entre dos dispositivos conectados a la red.

La arquitectura de nuestro sistema de red depende del tipo de aplicación que queramos. Existen diferentes tecnologías que nos permiten desarrollar entornos centralizados o descentralizados, y con diferentes niveles de control y eficiencia. Analizaremos los principales sistemas y cómo se programa cada uno de ellos utilizando una librería de red.

Finalmente, hablaremos sobre el fenómeno de los juegos en línea persistentes, que han cambiado tanto la forma de desarrollo de los juegos como el modelo de negocio asociado a ellos. En este contexto estudiaremos cómo se estructura la gestión de un juego persistente y cómo queda la información del usuario almacenada en servidores.

## 3.1. Historia de los juegos de red

Los primeros juegos considerados multijugador aparecieron en las universidades americanas a principios de los años ochenta. Se trataba de los MUD, Multi-User Dungeons, un tipo de juego basado en texto en el que te conectabas a un servidor mediante acceso a través de una terminal. El funcionamiento de estos juegos era parecido a lo que comúnmente conocemos como juegos de rol, donde el jugador se crea un avatar, y mediante la exploración del mundo tiene que desarrollar sus estadísticas.

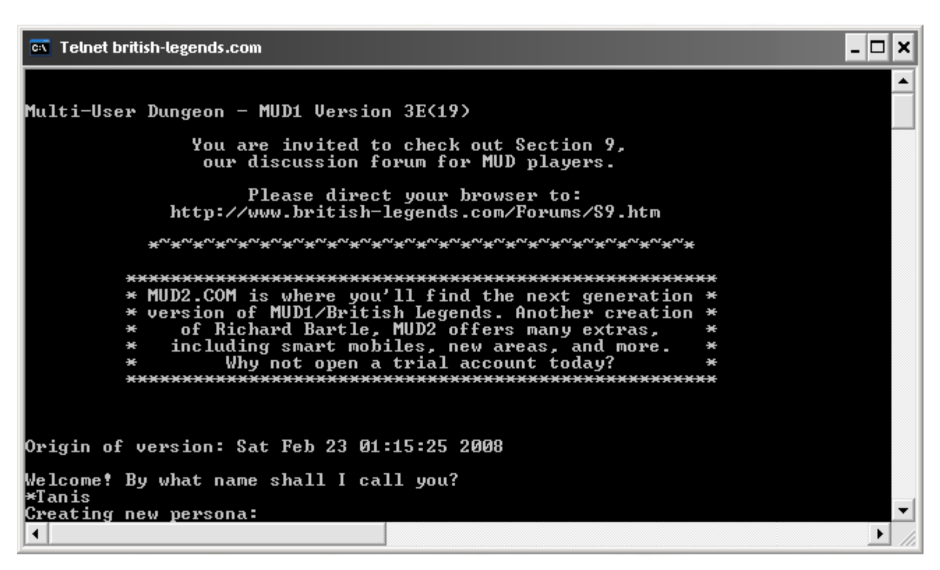



Figura 13. Consola del MUD British Legends

A principio de los años noventa apareció la posibilidad de utilizar las conexiones de red de los ordenadores personales para proporcionar un nuevo tipo de videojuegos. En esta época el acceso a la red se realizaba mediante el uso de módems o a través de una LAN (red de área local) mediante el protocolo IPX de Novell, lo que limitaba la cantidad de información que se podía transferir entre ordenadores.

**Protocolo IPX**

El protocolo IPX (Internet Packet eXchange) fue uno de los primeros que se utilizó para interconectar ordenadores en una red de área local. Al final acabó desapareciendo porque no era un protocolo óptimo para comunicar un gran número de ordenadores, al contrario que TCP/IP.

Los dos primeros juegos que sacaron el máximo partido a las capacidades multijugador fueron el Doom de Id Software, como pionero en los FPS, y el Warcraft de Blizzard, como pionero en juegos de estrategia en tiempo real. Ambos juegos permitían crear combates entre varios jugadores simultáneos, lo que se pasó a conocer comúnmente como un *deathmatch*.




Figura 14. Doom y Warcraft © Id Software y Blizzard respectivamente

El siguiente paso en la evolución de los juegos de red se produjo con la aparición de los primeros juegos en línea persistentes alrededor de 1995. Estos juegos combinaban parte de los dos puntos anteriores: tenían un servidor central donde los usuarios se conectaban para jugar, y, además, disponían de un

cliente gráfico que permitía interactuar de forma intuitiva con el entorno y los otros jugadores. Este tipo de juegos fueron llamados Massively Multiplayer Games (Juegos Mulitjugador Masivos).

Este tipo de juegos no pueden jugarse de forma individual, teniéndose que conectar a un servidor cada vez que queramos jugar una partida. Por esta razón, los juegos persistentes introdujeron un nuevo modelo de negocio para los videojuegos basado en subscripciones mensuales (para poder pagar el coste de mantenimiento de los servidores). La gran mayoría de estos juegos pertenecen al género de los MMORPG (*massively multiuser online role playing game*), y como ejemplos paradigmáticos podemos destacar el Ultima Online, que fue el primero en establecerse y el que más tiempo lleva en el mercado, y el World of Warcraft, uno de los últimos en llegar, pero que ha batido todos los récords sobre número de usuarios suscritos.




Figura 15. Ultima Online y World of Warcraft

En cuanto a las consolas, hubo algunas aproximaciones para añadir componentes hardware de red, pero la introducción a gran escala de las capacidades en línea se ha realizado principalmente en la última generación. Por un lado, las tres grandes compañías (Nintendo, Sony y Microsoft) han añadido algún tipo de hardware de red a sus videoconsolas. Y por otro lado, los estudios han comprobado que el desarrollo de juegos con opciones en línea mejora las ventas de los mismos, así que se ha potenciado la programación de la red en la mayoría de sus juegos (anteriormente, se dejaba como una opción muy remota en la que se invertía muy poco tiempo y dinero).

> Deberíamos puntualizar que la primera aproximación de las videoconsolas en el mundo de los videojuegos multiplayer se produjo con la introducción de la consola portátil GameBoy. La consola llevaba incorporado un puerto serie que permitía conectar hasta 4 dispositivos, y algunos de los juegos más vendidos (como por ejemplo el Tetris), ofrecían la posibilidad de competir con otros usuarios.

Paralelamente a los juegos, las tres compañías han creado unos entornos sociales donde se facilita la interactividad entre jugadores, haciendo mucho más fácil e intuitiva la transición hacia los juegos en línea. Estos entornos también han supuesto un nuevo modelo de negocio, donde se pueden adquirir direc-

tamente los juegos sin necesidad de soporte físico, además de servir como plataforma para poder lanzar más fácilmente juegos de desarrolladores independientes.




Figura 16. Servicios de suscripción en línea de Microsot (derecha) y Playstation (izquierda)

**ARPAnet**

ARPAnet se considera la primera red de transferencia de paquetes de la historia, y el punto de partida de lo que hoy conocemos por Internet. Fue desarrollada por el Departamento de Defensa de Estados Unidos en los años sesenta.

## 3.2. La arquitectura de Internet

La necesidad de comunicar varios dispositivos electrónicos nació paralelamente a éstos. Proyectos como ARPAnet crearon las primeras redes de ordenadores a pequeña escala, pero el gran salto cualitativo se produjo con la aparición de Internet como una red de comunicación a escala global. El éxito de Internet reside en que proporciona una infraestructura que permite que un gran número de dispositivos completamente diferentes puedan interactuar entre sí e intercambiar servicios e información de una manera segura y rápida.

### 3.2.1. Nociones sobre TCP/IP

La comunicación entre dos dispositivos conectados mediante una red se realiza a través de lo que conocemos como un protocolo de comunicación, es decir, un conjunto de normas que definen la "lengua" en la que hablan dos ordenadores entre sí. El conjunto de protocolos más extendido en la actualidad y el que se utiliza actualmente en Internet es conocido por TCP/IP (*transmission control protocol* / *Internet protocol*).

#### TCP/IP

El modelo TCP/IP está basado en una arquitectura de cinco capas que realizan funciones específicas dentro del proceso de comunicación entre dos aplicaciones, que son:

- **Capa física**. Se trata de la capa de más bajo nivel, donde se codifica la información por medio de señales eléctricas u ondas de radio y se transmite entre dos máquinas por un medio físico.

- **Capa de acceso al medio**. Se define el hardware y el protocolo de transmisión. Cada implementación usa un estándar de la industria. Por ejemplo: Ethernet, RDSI, Coaxial, Wireless...

- **Capa de red**. Se define la forma de comunicar varios puntos de una red. Se utiliza un protocolo sencillo de transmisión de paquetes llamado Internet Protocol o IP.

- **Capa de transporte**. Se define la forma de dividir y reordenar un flujo continuo de información en paquetes IP. También es responsable de asegurar la ausencia de errores. Hay dos protocolos en esta capa: *transport control protocol* (TCP) y *user datagram protocol* (UDP). Los analizaremos posteriormente con más detalle.

- **Capa de aplicación**. En esta capa se definen los protocolos de las aplicaciones, por ejemplo: SMTP, POP3, IMAP para el correo electrónico, HTTP para contenidos web, etc. Para una aplicación propia es necesario diseñar nuestro propio protocolo de comunicación.

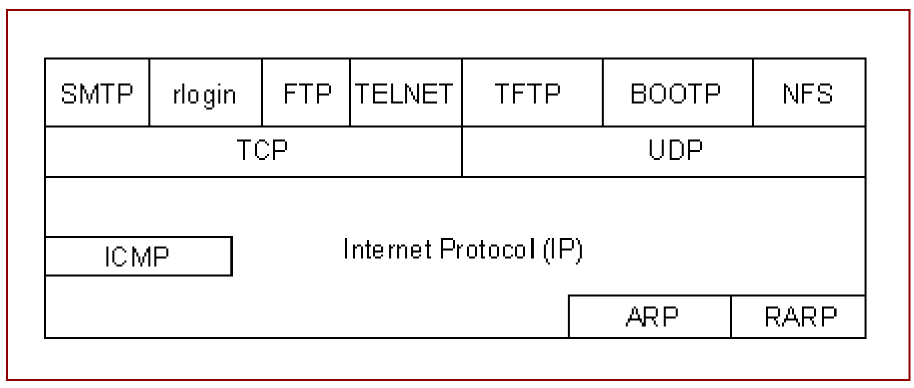



Figura 17. Ejemplo de algunos protocolos de las tres capas superiores de Internet

Vamos a analizar cómo se realiza una transmisión de información entre dos ordenadores mediante la arquitectura TCP/IP.

- Cuando una aplicación quiere enviar datos o información a otra, se divide la información en paquetes de datos. El tamaño de estos paquetes viene prefijado por el sistema operativo, normalmente son bastante pequeños (unos 1.500 bytes por paquete).

- Todos los paquetes tienen que descender todas las capas de la arquitectura TCP/IP dentro del ordenador. Cada una de estas capas añade información extra para que se pueda interpretar a qué ordenador va dirigido, y, una vez en la capa física, se realiza la comunicación real con el otro ordenador. Esta comunicación puede pasar por nodos intermedios.

- Una vez el destinatario recibe el paquete, éste va subiendo por la arquitectura usando la información contenida en los datos adicionales hasta que se entrega a la aplicación correspondiente.

- El destinatario tiene que recoger todos los paquetes enviados y ensamblarlos de nuevo para poder recuperar la información original.

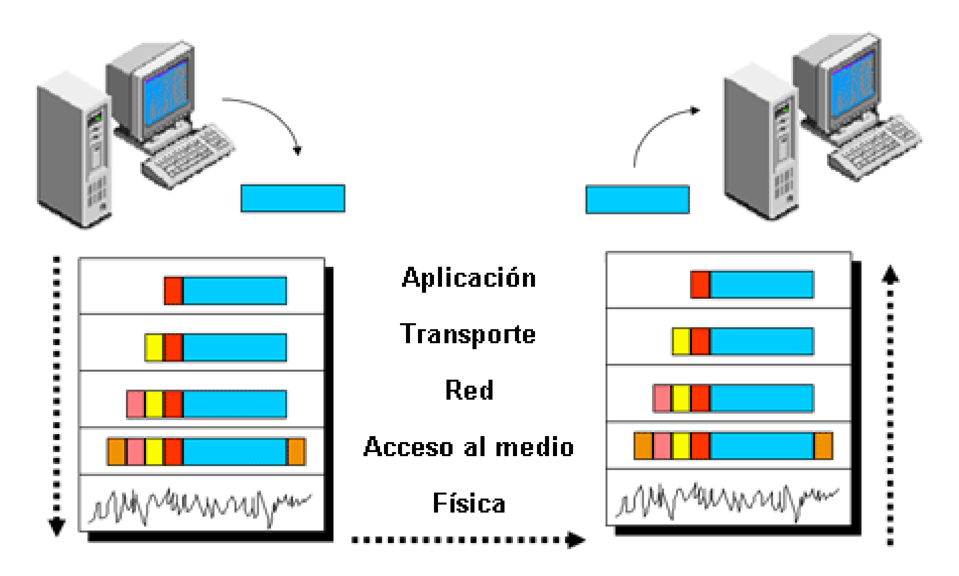



Figura 18. Cómo se transmite la información entre dos ordenadores

### Direccionamiento

Hemos visto que en cada una de las capas se añade información extra que nos ayuda a controlar la transmisión de los paquetes. Uno de los elementos más importantes que se añade en todos los niveles es la dirección, que permite identificar a qué aplicación y a qué máquina queremos enviar la información. En el modelo TCP/IP existe una dirección para cada nivel:

- **Capa de acceso al medio**. La dirección identifica el número MAC de la tarjeta de red que interviene en la comunicación. Se utiliza para saber a qué máquina de nuestra red local debemos enviar los datos para que se acerquen a su destino.

**Dirección MAC**

Un PC puede tener más de una dirección MAC si tiene más de una tarjeta de red, así como varias direcciones IP.

- **Capa de red**. En esta capa se añade la dirección IP, que identifica la localización exacta del destino. La dirección IP consiste en un número de 32 bits, que normalmente se representa mediante cuatro números decimales de 8 bits separados por puntos.

- **Capa de transporte**. La dirección nos permite saber a qué aplicación se tiene que entregar la información. Esta dirección se representa con un número de 16 bits que se llama puerto.

- **Capa de aplicación**. No hay ningún estándar y es la propia aplicación la que guarda y mantiene a qué componente se tiene que distribuir los datos.

### Tipos de comunicación

El protocolo de comunicación puede ser orientado a la conexión, en el caso de que se requiera la existencia de un canal persistente entre los dos agentes por donde transmitimos la información, o no orientado a la conexión si no se requiere este canal para transferir los datos.

- La **comunicación orientada a la conexión** requiere el establecimiento de un canal de datos mediante una conexión inicial entre los dos agentes que intervienen en la comunicación. Al principio, los dos agentes negocian entre ellos y crean una conexión dedicada para poder hablar entre sí. Una vez se establece el canal, los dos agentes envían y reciben los paquetes de datos a través de éste. En la arquitectura TCP/IP este tipo de conexión se realiza mediante el uso del protocolo TCP. Algunos ejemplos de protocolos a nivel de aplicación que trabajan con él son HTTP, POP3, FTP...

- La **comunicación no orientada a la conexión** permite enviar paquetes entre dos agentes sin establecer ningún tipo de conexión inicial. En este caso, el protocolo que se utiliza es el UDP y como ejemplos de aplicaciones basadas en este tipo de conexión, tenemos varios sistemas de transferencia de vídeo o audio, DHCP, DNS...

Vamos a comparar los dos protocolos con más detalle.

| TCP | UDP |
|---|---|
| Necesita establecer un canal de comunicación. | No necesita de ningún tipo de conexión previa para poder enviar paquetes. |
| Garantiza que los datos se han transferido íntegramente entre los dos agentes. | No garantiza que los datos se reciban. |
| Los paquetes se reciben en el mismo orden en que son enviados. | Tampoco garantiza que se reciban en el mismo orden en que se han enviado. |
| Los paquetes con los datos pueden tener un tamaño de bytes variable. | Los paquetes (en este caso se llaman datagramas) de datos tienen todos la misma longitud. |
| Si hay algún problema en la transmisión, los paquetes se reenvían, creando más tráfico en la red. | No existe el reenvío de paquetes. |
| Es un protocolo lento, ya que tiene que garantizar que todos los datos se reciben y que están en orden. | Es un protocolo rápido, ya que no requiere comprobaciones adicionales. |

**¿Qué tipo de conexión debemos utilizar en un juego?**

La decisión depende de qué tipo de juego queremos implementar. Por ejemplo, en un juego de estrategia donde cada movimiento es clave para el funcionamiento del sistema, es preferible una conexión TCP que nos garantice que todas las acciones que tome el usuario se ejecuten. En cambio, en un FPS, donde se puede actualizar el sistema 60 veces por segundo, no podemos utilizar un protocolo TCP porque nos podría ralentizar el sistema, así que quizás la mejor opción es utilizar UDP y simplemente no preocuparnos por aquellos paquetes que se pierden.

Una buena alternativa es utilizar una solución mixta, es decir, podemos tener a la vez un canal de control por donde se transmita la información clave con una conexión TCP, y un canal de datos donde se transmita el resto de la información no crucial mediante el protocolo UDP. Un ejemplo práctico de qué tenemos que enviar en cada uno de estos canales vendría dado por la distancia de los elementos a nuestra posición. Por ejemplo, en un juego multiusuario podríamos enviar por TCP los datos referentes a aquellos otros usuarios que se encuentran cerca y con los que podemos llegar a interactuar, y por otro lado, enviar por UDP los datos de usuarios lejanos, con los que la posibilidad de interactuar sea casi nula.

### 3.3. Programación de la red

La programación de un juego de red requiere dos elementos imprescindibles. En primer lugar, necesitamos crear las conexiones entre los clientes y saber cómo enviar y gestionar la información. Y en segundo lugar, necesitamos diseñar un protocolo que defina cómo se comunican nuestras dos aplicaciones entre sí, es decir, tenemos que crear un lenguaje que permita describir los estados de nuestro juego y de los elementos que lo componen.

La programación se puede efectuar de distintas maneras, según el control que queramos tener de lo que se envía y se recibe:

- La primera opción es utilizar las librerías de *sockets*, que nos proporcionan un acceso de bajo nivel a la red. En este sentido, tenemos un control total de cada byte de información que se transmite por la red, pero esto también implica que tenemos que encargarnos de cualquier problema que pueda existir, además de gestionar casos como múltiples conexiones que requieren estructuras más complejas.

- Otra opción es utilizar una librería de más alto nivel, que gestione todas las conexiones concurrentes y el estado de las mismas. En este caso, tenemos menos control, pero si la librería está implementada de forma eficiente vamos a reducir mucho el trabajo necesario para poder implementar las conexiones, y nos podremos centrar en el diseño del protocolo.

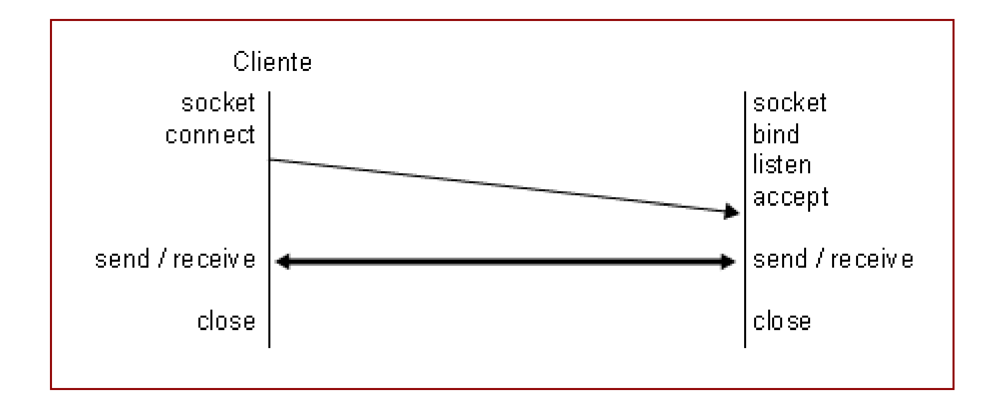
El diseño del protocolo y la optimización del mismo lo trataremos en un apartado posterior.

#### 3.3.1. Programación de red de bajo nivel: Las librerías de *sockets*

La librería básica para programar en red se basa en el concepto de *socket*. Un *socket* es una entidad que permite que una computadora intercambie datos con otras computadoras. Un *socket* identifica a una conexión particular entre dos ordenadores y se compone siempre de cinco elementos: una dirección de red (identificador IP), un puerto de la máquina origen, una dirección de red, un puerto de la máquina destino y el tipo de protocolo (TCP, UDP...).

El concepto de *socket* se desarrolló en la Universidad de Berkeley cuando empezaron a comunicar varios ordenadores Unix. A partir de esa experiencia, se desarrolló una librería sencilla que permitía aprovechar este trabajo y escribir nuevos programas capaces de comunicarse entre ellos.

El cronograma de las funciones que se llaman durante una comunicación se puede representar con la siguiente gráfica:




En el siguiente ejemplo creamos un programa que actúa de cliente pidiéndole una página web a un servidor HTTP. Utilizaremos la API de *socket* de Windows y, para ello, se tendrá que enlazar el código con la librería "ws2_32.lib".

**uocfox.cpp**

```
#include <winsock2.h>
#include <windows.h>
#include <iostream>

int main(int argc, char* argv[])
{
   if (argc != 2)
   {
      std::cerr << argv[0] << " <hostname>" << std::endl;
      exit(-1);
   }
...
```

En primer lugar tenemos que inicializar la librería de *sockets* para poder tener acceso a la interfaz de red.

```
...
   int result;
   WSADATA wsaData;
   result = WSAStartup(MAKEWORD(2,2), &wsaData);
   if (result != NO_ERROR)
   {
      std::cerr << "WinSocket Error: " << WSAGetLastError() << std::endl;
      WSACleanup();
      exit(-1);
   }
```

```
...
```

Una vez tenemos acceso a la red, abrimos un nuevo *socket* por el que vamos a enviar y recibir los datos. En este ejemplo empezamos por abrir un *socket* TCP, llenamos toda la información que nos interesa de la conexión en la variable sockaddr_in addr e intentamos conectarnos con el ordenador destino para establecer el canal persistente.

```
...
   SOCKET client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (client == INVALID_SOCKET)
   {
      std::cerr << "socket error: " << WSAGetLastError() << std::endl;
      WSACleanup();
      exit(-1);
   }

   hostent* he = gethostbyname(argv[1]);
   sockaddr_in addr;
   addr.sin_family = AF_INET;
   memcpy(&addr.sin_addr.s_addr, he->h_addr, he->h_length);
   addr.sin_port = htons(80);

   result = connect(client, (sockaddr*) &addr, sizeof(addr));
   if (result == SOCKET_ERROR)
   {
      std::cerr << "connect error: " << WSAGetLastError() << std::endl;
      WSACleanup();
      exit(-1);
   }
...
```

Cuando tenemos el canal abierto, podemos enviar y recibir datos de él. En este caso, como queremos pedir una página web, enviamos un comando HTTP y esperamos como respuesta una página web. Más adelante, veremos qué tipo de información enviamos cuando queremos conectar dos juegos.

```
...
   //Enviar mensaje
   const char* msg = "GET / HTTP/1.0\r\n\r\n";
   result = send(client, msg, (int) strlen(msg), 0);
   if (result == SOCKET_ERROR)
   {
      std::cerr << "send error: " << WSAGetLastError() << std::endl;
      WSACleanup();
      exit(-1);
   }
```

```
      shutdown(client, SD_SEND);
      //Esperar respuesta
      do
      {
         char buf[1024];
         result = recv(client, buf, sizeof(buf) - 1, 0);
         if (result == 0)
         {
            std::cerr << "server closed the connection" << std::endl;
         }
         else if (result > 0)
         {
            buf[result] = '\0';
            std::cout << buf;
         }
         else
         {
            std::cerr << "recv error: " << WSAGetLastError() << std::endl;
            WSACleanup();
            exit(-1);
         }
      } while (result > 0);
      closesocket(client);

      WSACleanup();
      return 0;
   }
```

Para trabajar de manera eficiente con *sockets*, también es necesario trabajar con multihilos, ya que en una comunicación es necesario hacer esperas para recibir información del otro nodo (servidor o cliente) y este tiempo es crucial en otros aspectos del videojuego.

El sistema de multihilos está muy relacionado con el sistema operativo. Por ejemplo, en Windows debemos trabajar con su propia API de mulithilos y en entornos UNIX con POSIX. En este punto nos es necesario buscar otra librería que nos abstraiga el sistema operativo para hacer nuestro código más portable como, por ejemplo, Boost o SDL.

**Sistema de multihilos**

El uso de varios hilos en un videojuego nos permite crear diferentes tareas que están trabajando todo el tiempo. En el caso de ordenadores o consolas con varias CPU la programación multihilos nos permite sacar el máximo partido a todas ellas.

Otro punto importante a tener en cuenta es que, en varios tipos de juegos, el usuario demanda poder hablar por micrófono con los demás jugadores. Para implementar este punto con eficiencia, debemos dedicar muchos recursos y tener un amplio conocimiento de redes y transferencia de sonido (*streaming*) en tiempo real.

En este punto, volvemos a estudiar si nos es más necesario hacer el código usando las librerías básicas (*sockets* y API de multihilos) o bien trabajar con librerías de terceros que nos abstraigan estas técnicas.

### 3.3.2. API de programación de red

Tenemos varias librerías que podemos utilizar para programar la parte de red sin necesidad de trabajar a tan bajo nivel como se necesita con *sockets*. Las que consideramos más relevantes son:

- **DirectPlay**. Librería que propone Microsoft dentro de su conjunto de librerías DirectX. Soporta la creación de sistemas cliente/servidor y *peer-to-peer*. La comunicación se realiza tanto por conexiones TCP como UDP. Implementa sistemas para conectar usuarios a través de NAT (*network address translation*) y transmisión de voz. Como todos los componentes de DirectX, Microsoft proporciona el código necesario para enlazarlo con nuestro código de manera gratuita. Está disponible para sistemas Microsoft Windows y Xbox.

- **Raknet**. Librería que nos permite trabajar con sistemas cliente/servidor y *peer-to-peer*. Trabaja con mensajes UDP. Tiene soporte para trabajar en la transmisión de voz con la librería de sonido FMOD. En el caso de crear una aplicación comercial, será necesario comprar una licencia de esta librería. Está portada a sistemas Windows, Linux y consolas.

- **SDL_Net**. Librería que crea una interfaz común para varios sistemas. Esta interfaz permite trabajar con *sockets* en Windows y en Unix. Está preparada para trabajar con multihilos usando la misma librería SDL. No ofrece sistemas para la transmisión de voz.

- **HawkNL**. Librería libre que permite conexiones TCP y UDP. La librería es muy sencilla y suele utilizarse con otra llamada HawkVoice, que permite la transmisión de voz. Está disponible para sistemas Unix y Windows.

Hemos decidido utilizar la librería Raknet para realizar los ejemplos de este módulo por su facilidad de uso y por las funcionalidades que proporciona. Además, se trata de una librería suficientemente consolidada, ya que se ha utilizado para crear juegos comerciales y es gratuita para desarrollos no comerciales.

Presentamos en el siguiente ejemplo un código sencillo basado en el tutorial de la librería para conectar dos procesos usando Raknet. En primer lugar tenemos que incluir todos los ficheros externos que vamos a utilizar. Dependiendo del tipo de funciones que queramos llamar, tendremos que incluir más o menos ficheros.

El código nos permite crear un chat entre dos usuarios. Dependiendo de los parámetros que pasemos al código principal nos creara un servidor de chat o un cliente de chat. En el ejemplo, usaremos la técnica de RPC (*remote procedure call*, o llamada a procedimiento remoto).

***Remote procedure call***

Una llamada a procedimiento remoto es una técnica que se utiliza para que un proceso pueda ejecutar una parte de código de otro y esperar su respuesta. Es una capa que se programa por encima de una librería de red y evita los problemas más comunes (sincronización, envío de información...).

Hay muchos tipos de implementaciones de RPC, pero en el caso concreto de la librería RakNet, han preferido crear una interfaz muy sencilla maximizando la efectividad.

### uocchat.cpp

```
#include <cstdlib>
#include <iostream>
#include <sstream>

#include <conio.h> // simplifica la lectura del teclado
#include <Windows.h>
#include <RakNetworkFactory.h>
#include <RakPeerInterface.h>
#include <MessageIdentifiers.h>

void showError(char* argv[])
{
   std::cerr << "Error:" << std::endl;
   std::cerr << " " << argv[0] << " server port" << std::endl;
   std::cerr << " " << argv[0] << " client servername port" << std::endl;
}

int main(int argc, char* argv[])
{
   switch(argc)
   {
   case 1:
   case 2:
      showError(argv);
      exit(-1);
   case 3:
      if (strcmp(argv[1], "server") != 0)
      {
         showError(argv);
         exit(-1);
      }
      startServer(argv, argv[2]);
      break;
```

```
      case 4:
         if (strcmp(argv[1], "client") != 0)
         {
            showError(argv);
            exit(-1);
         }
         startClient(argv, argv[2], argv[3]);
         break;
      default:
         showError(argv);
         exit(-1);
      }
      return 0;
   }
   ...
```

La primera función que desarrollamos es la del servidor de chat. Al igual que con los *sockets*, lo primero que tenemos que hacer es abrir una conexión por donde recibiremos todas las líneas de chat de los usuarios que después redistribuiremos entre ellos.

```
   void startServer(char *argv[], const char *sport)
   {
      const int MAX_CLIENTS = 30;

      unsigned short port;
      if (! (std::stringstream(sport) >> port))
      {
            showError(argv);
            exit(-1);
      }
      std::cout << "Start server at port " << port << std::endl;
      RakNet::RakPeerInterface *peer = RakNet::RakPeerInterface::GetInstance();

      RakNet::SocketDescriptor socket(port, 0);
      peer->Startup(MAX_CLIENTS, &socket, 1);
      peer->SetMaximumIncomingConnections(MAX_CLIENTS);
   ...
```

Una vez tenemos la interfaz creada para poder recibir paquetes, lo siguiente es poner el código para que espere a que se reciban paquetes y poderlos tratar. Hay dos maneras de ver si hay paquetes:

- **De forma asíncrona**. Simplemente miramos cada cierto tiempo si ha llegado un paquete. Si no ha llegado ninguno, seguimos ejecutando nuestro código y al cabo de un rato volvemos a mirar si ha llegado algo. Esto nos

permite no tener que dejar el código pendiente de si llega o no la información.

- **De forma síncrona**. En este caso cuando miramos si ha llegado un paquete, bloqueamos el sistema y esperamos hasta que llegue el primer paquete. El programa no continua ejecutándose a no ser que tengamos un sistema multihilos como el que hemos comentado anteriormente.

En el siguiente código podemos ver que se trata de un sistema asíncrono. Una vez que recibimos el paquete, lo primero que miramos es de qué tipo de paquete se trata, utilizando los diferentes tipos que tiene por defecto RakNet. En el caso de un paquete de datos, se ejecuta el caso *default*, y aquí es donde nosotros haremos lo que queramos con la información recibida de uno de los clientes.

```
...
   char message[2048];
   while (true)
   {
   RakSleep(100);
   for (RakNet::Packet* p = peer->Receive();
         p;
         peer->DeallocatePacket(p), p = peer->Receive()) {
   unsigned char packetIdentifier = GetPacketIdentifier(p);
   switch (packetIdentifier)
   {
      case ID_DISCONNECTION_NOTIFICATION:
         std::cout << "ID_DISCONNECTION_NOTIFICATION from "
                   << p->systemAddress.ToString(true) << std::endl;
         break;

      case ID_NEW_INCOMING_CONNECTION:
         std::cout << "ID_NEW_INCOMING_CONNECTION from "
                   << p->systemAddress.ToString(true)
                   << " with GUID "
                   << p->guid.ToString() << std::endl;
       break;

      case ID_INCOMPATIBLE_PROTOCOL_VERSION:
         std::cout << "ID_INCOMPATIBLE_PROTOCOL_VERSION" << std::endl;
         break;

      case ID_CONNECTED_PING:
      case ID_UNCONNECTED_PING:
         std::cout << "Ping from "
                   << p->systemAddress.ToString(true) << std::endl;
       break;
```

```
            case ID_CONNECTION_LOST:
               std::cout << "ID_CONNECTION_LOST from "
                         << "p->systemAddress.ToString(true) << std::endl;
               break;

            default:
               std::cout << p->data << std::endl;
               sprintf(message, "%s", p->data);
               peer->Send(message,
                          (const int) strlen(message)+1,
                           HIGH_PRIORITY, RELIABLE_ORDERED, 0, p->systemAddress, true);
               break;
            }
         }
      }
   }
```

En la segunda función del código implementamos la parte referente al cliente. En este caso abrimos un *socket* e intentamos conectarlo con el servidor que nos hayan introducido como parámetro.

```
void startClient(char *argv[], const char *servername, const char *sport)
{
   unsigned short port;
   if (! (std::stringstream(sport) >> port))
   {
      showError(argv);
      exit(-1);
   }

   std::cout << "Start client and connect to " << servername << " at "
             << port << std::endl;
   RakNet::RakPeerInterface *peer = RakNet::RakPeerInterface::GetInstance();
   RakNet::SocketDescriptor socket;
   peer->Startup(1, &socket, 1);
   peer->Connect(servername, port, 0, 0);
```

A partir de ahora, realizamos un bucle que nos permita ir leyendo del teclado y de la red.

```
...
   char message[2048];
   while (true)
   {
      RakSleep(100);
      if (kbhit())
```

```
        {
           Gets(message, sizeof(message));
           peer->Send(message,
           (int) strlen(message)+1,
           HIGH_PRIORITY, RELIABLE_ORDERED, 0,
           RakNet::UNASSIGNED_SYSTEM_ADDRESS, true);
        }

     for (RakNet::Packet* p = peer->Receive();
        p;
        peer->DeallocatePacket(p), p = peer->Receive()) {
        unsigned char packetIdentifier = GetPacketIdentifier(p);

        switch (packetIdentifier)
        {
        case ID_DISCONNECTION_NOTIFICATION:
           std::cout << "ID_DISCONNECTION_NOTIFICATION" << std::endl;
           break;
        case ID_ALREADY_CONNECTED:
           std::cout << "ID_ALREADY_CONNECTED" << std::endl;
           break;
        case ID_INCOMPATIBLE_PROTOCOL_VERSION:
           std::cout << "ID_INCOMPATIBLE_PROTOCOL_VERSION" << std::endl;
           break;
        case ID_REMOTE_DISCONNECTION_NOTIFICATION:
           std::cout << "ID_REMOTE_DISCONNECTION_NOTIFICATION" << std::endl;
           break;
        case ID_REMOTE_CONNECTION_LOST:
           std::cout << "ID_REMOTE_CONNECTION_LOST" << std::endl;
           break;
        case ID_REMOTE_NEW_INCOMING_CONNECTION:
           std::cout << "ID_REMOTE_NEW_INCOMING_CONNECTION" << std::endl;
           break;
        case ID_CONNECTION_BANNED:
           std::cout << "We are banned from this server." << std::endl;
           break;
        case ID_CONNECTION_ATTEMPT_FAILED:
           std::cout << "Connection attempt failed" << std::endl;
           break;
        case ID_NO_FREE_INCOMING_CONNECTIONS:
           std::cout << "ID_NO_FREE_INCOMING_CONNECTIONS" << std::endl;
           break;
        case ID_INVALID_PASSWORD:
           std::cout << "ID_INVALID_PASSWORD" << std::endl;
           break;
        case ID_CONNECTION_LOST:
           std::cout << "ID_CONNECTION_LOST" << std::endl;
```

```
            break;
         case ID_CONNECTION_REQUEST_ACCEPTED:
            std::cout << "ID_CONNECTION_REQUEST_ACCEPTED to "
                      << p->systemAddress.ToString(true)
                      << " with GUID "
                      << p->guid.ToString() << std::endl;
            break;
         case ID_CONNECTED_PING:
         case ID_UNCONNECTED_PING:
            std::cout << "Ping from "
                      << p->systemAddress.ToString(true) << std::endl;
            break;
         default:
            std::cout << p->data << std::endl;
            break;
         }
      }
   }
}
```

Para poder compilar el código, necesitaremos también de la siguiente función:

```
unsigned char GetPacketIdentifier(RakNet::Packet *p)
{
   if (p==0)
      return 255;

   if ((unsigned char)p->data[0] == ID_TIMESTAMP)
   {
      RakAssert(p->length >
            sizeof(RakNet::MessageID) + sizeof(RakNet::Time));
      return (unsigned char) p->data[sizeof(RakNet::MessageID)
            + sizeof(RakNet::Time)];
   }
   else
      return (unsigned char) p->data[0];
}
```

Si abrimos tres instancias de la aplicación (uno como servidor y dos como clientes), obtenemos la siguiente respuesta:

```
uocchat client localhost 5555
Start client and connect to localhost at 5555
Our connection request has been accepted.
hola
111
1
```

```
hola
11
222
1
1
22
```

**uocchat client localhost**

```
Start client and connect to localhost at 5555
Our connection request has been accepted.
hola
11
222
1
1
22
```

**uocchat servidor 5555**

```
Start server at port 5555
A connection is incoming.
hola
111
1
A connection is incoming.
hola
11
222
1
1
22
A client lost the connection.
A client lost the connection.
```

### 3.4. Arquitecturas de comunicación

Hay diferentes maneras de conectar los distintos agentes. La forma más clásica de separar las conexiones es utilizando el rol que tiene cada uno de los agentes que intervienen en la comunicación. En el caso de que queramos una jerarquía de comunicación, necesitaremos un sistema cliente servidor, y en el caso de que queramos que todos los agentes se comporten por igual, necesitaremos una arquitectura *peer-to-peer*.

### 3.4.1. Sistemas cliente/servidor

En este tipo de comunicación definiremos a un agente como servidor y a otro como cliente:

- Por un lado, el **servidor** es aquel que permanece a la espera de peticiones de clientes y va sirviendo las diferentes peticiones a medida que van llegando.

- Por otro lado, el **cliente** es responsable de realizar una petición a un servidor.

Por ejemplo, cada vez que un cliente quiere enviar un mensaje o alguna información a los otros clientes, esta información la envía al servidor y el servidor es el que la distribuye entre aquellos otros clientes a los que vaya redirigida.

Éstas son algunas **ventajas** que obtenemos al utilizar un sistema cliente servidor:

- La seguridad se puede controlar más, ya que el servidor puede detectar si alguno de los clientes está realizando alguna acción no apropiada.

- En el caso de que haya datos persistentes, permite garantizar su integridad.

- Normalmente, los servidores son sistemas dedicados con un hardware específico, lo que aumenta su rendimiento frente a ordenadores estándares.

- Es mucho más fácil gestionar la gestión de usuarios y de partidas.

Por otro lado, éstos son algunos de los **inconvenientes** asociados al uso de un sistema cliente/servidor:

- Necesitamos crear dos tipos diferentes de programas, uno que se encargue en el servidor de la gestión del juego y otro en el del cliente, que nos permita jugarlo.

- Si queremos utilizar un servidor dedicado, el hardware nos puede salir bastante caro, además, necesitaremos un administrador de red que nos ayude a configurar y mantener el sistema para su buen funcionamiento.

- En el caso de que haya un problema en el servidor, se interrumpe completamente el desarrollo del juego. Además, la conexión del servidor puede suponer un cuello de botella en el rendimiento global del sistema.

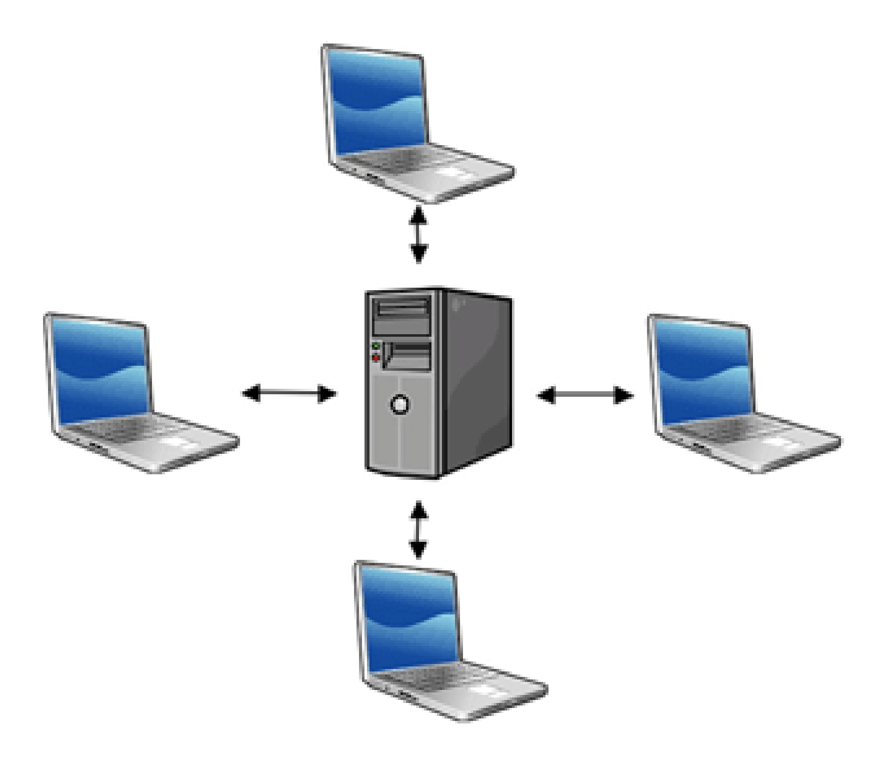


Figura 19. Esquema de un entorno cliente/servidor

En ejemplos anteriores sobre las librerías de *sockets* y la aplicación RakNet hemos visto la implementación de un sistema cliente servidor.

Otro aspecto adicional que es muy importante tener en cuenta cuando tratamos con un sistema basado en cliente/servidor es la gestión de las conexiones que el servidor tiene abiertas en todo momento.

En la librería RakNet hay una serie de paquetes virtuales que nos informan de cambios en la lista de clientes. Estos mensajes los dividimos en dos grupos:

- **Mensajes de servidor**
  - ID_REMOTE_DISCONNECTION_NOTIFICATION: Un cliente se ha desconectado.
  - ID_REMOTE_CONNECTION_LOST: Se ha perdido la conexión con un cliente.
  - ID_REMOTE_NEW_INCOMING_CONNECTION: Un cliente se quiere conectar.

- **Mensajes de cliente**
  - ID_CONNECTION_REQUEST_ACCEPTED: El servidor ha aceptado nuestra conexión.
  - ID_NEW_INCOMING_CONNECTION: Un cliente se quiere conectar.
  - ID_NO_FREE_INCOMING_CONNECTIONS: No hay más conexiones disponibles en el servidor al que queremos conectar.
  - ID_DISCONNECTION_NOTIFICATION: Nos hemos desconectado del servidor.

– ID_CONNECTION_LOST: Hemos perdido la conexión.

Un servidor obtiene la lista de conexiones de sus clientes a través del método GetConnectionList de su objeto *peer*.

Es posible que necesitemos organizar nosotros mismos esta lista. Por ejemplo, podemos mantener tantas listas como equipos de jugadores que están participando en el juego. En este caso, por ejemplo, cada jugador al conectarse deberá enviar un mensaje indicando a qué equipo pertenece y el servidor deberá mantener estas conexiones.

### 3.4.2. Sistemas *peer-to-peer*

En un sistema *peer-to-peer* todos los agentes se encuentran en el mismo nivel jerárquico, es decir, consideramos a todos los elementos como iguales. En un juego basado en un sistema *peer-to-peer* no hay ningún ordenador que tenga más control del juego que los otros, ni existe ningún mediador que nos sirva para distribuir el estado del juego o para enviar mensajes entre los diferentes clientes.

**Sistemas *peer to peer***

Los sistemas *peer to peer* son populares para juegos multiplayer en entornos LAN, principalmente porque es más fácil realizar un *broadcast* de la información, es decir, enviar la información a todos los otros *peers* a la vez.

Algunas de las **ventajas** que proporciona un entorno *peer-to-peer* son las siguientes:

- No es necesario invertir en servidores específicos ni en administradores, ya que cada usuario actúa como administrador de su propio equipo.

- Se desarrolla un solo cliente común para todos, lo que facilita la programación del juego y el diseño de su arquitectura interna.

- No se depende de un sistema central, lo que evita problemas de conexiones y cuellos de botella. Si un cliente no funciona, esto no repercute en el resto de jugadores.

Por otro lado, los sistemas *peer-to-peer* tienen los siguientes **inconvenientes**:

- Son poco escalables. Se pueden utilizar sin problemas a pequeña escala, pero si intentamos utilizarlos con muchos clientes y distribuidos por todo el planeta, el rendimiento acostumbra a ser muy bajo.

- La falta de un sistema central hace que las posibilidades de desincronización entre clientes sea mucho mayor, provocando posibles incongruencias en lo que dos clientes pueden estar viendo al mismo tiempo.

- La seguridad de este tipo de sistemas es muy baja, ya que tenemos que confiar en que los otros clientes sean fiables.

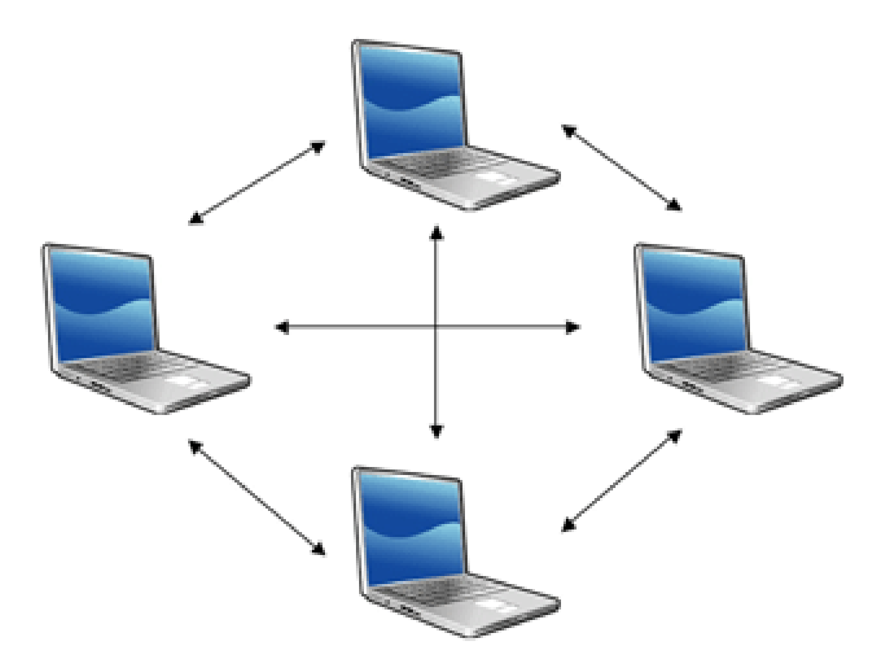


Figura 20. Conexión sistema *peer to peer*

Las primeras implementaciones de sistemas *peer-to-peer* utilizaban la técnica de *broadcast* (envío a todos los nodos) de las redes LAN. A medida que la tecnología de red ha ido evolucionando, estos sistemas han dejado de utilizarse básicamente porque en Internet es inviable hacer *broadcasting* de la misma manera como se hacía en las redes LAN.

Un sistema *peer-to-peer* se implementa creando varias conexiones cliente/servidor entre los componentes del juego (los *peers*). En el caso de crear todas las conexiones posibles entre los diferentes *peers*, se tendrían que hacer $N^2$ conexiones, lo que también es inviable y, por lo tanto, necesitamos generar estructuras que ahorren y compartan todas las conexiones posibles.

Actualmente existe mucha actividad en la investigación de sistemas *peer-to-peer*. La estructura más básica que permite mantener información con un acceso rápido son las DHT (*distributed hash table*, o tablas de dispersión distribuidas).

**DHT**

Un ejemplo de DHT es Chord.

Una estructura DHT permite:

- Mantener información usando pares clave–valor.
- Esta información se mantiene en alguna parte de la red de *peers*.
- Cada *peer* puede introducir o borrar una nueva clave con su valor en la red.
- Cada *peer* puede consultar el valor de una clave. Por ejemplo, Chord asegura que el número de saltos (peticiones entre *peers*) para obtener un valor es de log N.

**Documentación**

Para profundizar sobre este tema, os remitimos a la conferencia NETGAMES, donde, entre otros, se trata con detalle la problemática de este punto.

### 3.4.3. Sistemas híbridos

Una tercera opción de la que disponemos es utilizar parte de las dos arquitecturas descritas previamente, combinando clientes, servidores y *peers*. Aprovecharemos las ventajas de cada tipo de arquitectura para realizar cierto tipo de funciones concretas:

- Por un lado, podemos utilizar las conexiones cliente/servidor para aquellos momentos en que necesitemos cierto orden, como iniciar la partida o comprobar cada cierto tiempo que todos los clientes se encuentran en el mismo estado.
- Por otro lado, podemos utilizar conexiones *peer-to-peer* para distribuir los cambios del cliente de forma más frecuente y rápida, o para enviar otro tipo de información que no es necesario sincronizar (como por ejemplo un chat).

Un problema particular en un juego, basado en una arquitectura *peer-to-peer* y donde existen elementos controlados mediante inteligencia artificial, es saber cuál de los clientes toma las decisiones de IA. Una posibilidad es utilizar el servidor como un *peer* más que se encarga de mover todos los elementos no controlados por los jugadores, convirtiéndose en un jugador más dentro de la red *peer-to-peer*.

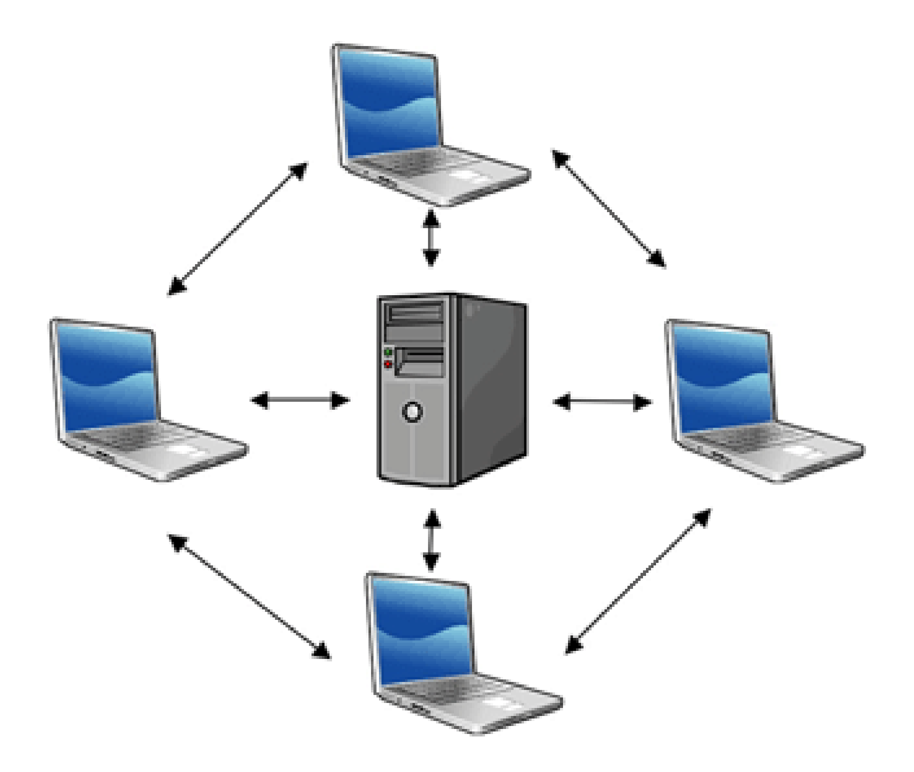


Figura 21. Ejemplo de una arquitectura híbrida

### 3.5. El protocolo de comunicación

Una vez hemos establecido los canales para poder comunicar un grupo de aplicaciones, el paso siguiente es definir lo que vamos a enviar a través de estos canales.

En primer lugar, tenemos que definir el formato de nuestros paquetes de información, es decir, cómo va a estar organizada la información que vamos a enviar. Un paquete necesita normalmente una cabecera con unos contenidos mínimos para poder ser interpretado correctamente por parte del destinatario. Algunos de los campos comunes que podemos encontrar en un paquete de datos son los siguientes:

- **Timestamp**. Para poder sincronizar los eventos en los clientes y, además, poder saber si llega algún paquete fuera de tiempo (en el caso de utilizar UDP).

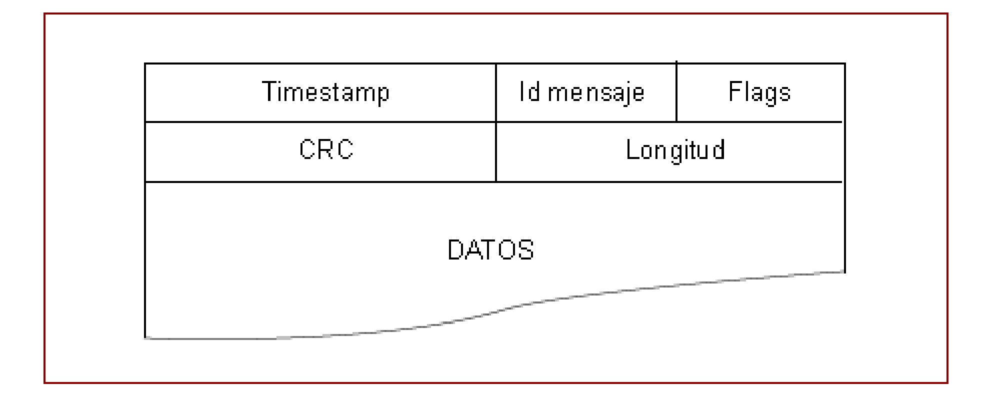
- **Identificador del mensaje**. Identificar qué tipo de datos están incluidos en el paquete.

- **Flags**. Bits adicionales que nos permitan saber más sobre estos datos rápidamente. Normalmente acostumbran a ser valores booleanos que nos permitan hacer un preproceso del paquete.

- **CRC**. Cyclic Redundance Check. Es un sistema para garantizar la integridad de la información contenida en el paquete.

- **Longitud del paquete**. Información sobre el número de bytes totales.

- **Datos**. El mensaje que queremos distribuir a los otros clientes.




Algunas consideraciones extra a tener en cuenta al diseñar el contenido de los paquetes de datos:

- Optimizar al máximo, compactar la información todo lo posible. Si se puede enviar un entero, no enviar un flotante ni una cadena de texto. Todos los datos booleanos se pueden compactar como *flags* en enteros.

- Enviar el mínimo de paquetes. Si tenemos que enviar la posición de diez elementos, enviar un solo paquete con todos los datos, en lugar de diez paquetes con los datos por separado. Es más rápido darle trabajo de codificación/decodificación a la CPU que lo que tarda en viajar la información.

- No enviar información no necesaria o redundante. Si por ejemplo no tenemos que enviar a cada momento la cantidad de vida del personaje, no la pongamos en todos los paquetes. Es mejor tener un paquete especial que solo se va a enviar si hay un cambio en el estado de esta variable, en lugar de enviar repetidamente el mismo valor.

Una vez tengamos diseñado nuestro formato de paquete, el último paso necesario para hacer funcionar nuestra aplicación en red es definir la secuencia de mensajes. Esta secuencia tiene que definir cómo iniciar una partida en red, cómo actualizar la información durante la partida y cómo finalizar la conexión. Una buena manera de resumir esta secuencia es mediante un diagrama de flujo que describa el tipo de mensaje que podemos recibir en cada momento y lo que tenemos que hacer con el mismo.

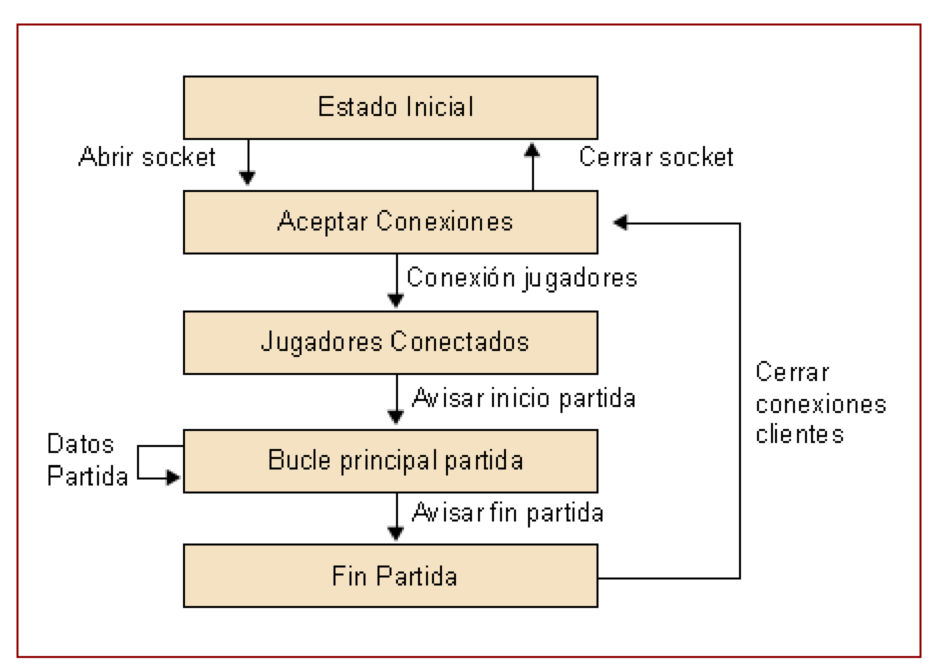



Figura 22. Ejemplo de diferentes estados donde es necesario algún tipo de paquete especial para poder cambiar al siguiente estado

Cada uno de estos estados puede a su vez incluir un diagrama de estados más detallado que describa las características de los paquetes que se pueden enviar dentro de este estado. Por ejemplo, en el estado de aceptar conexiones podríamos tener un diagrama dentro con los siguientes subestados.

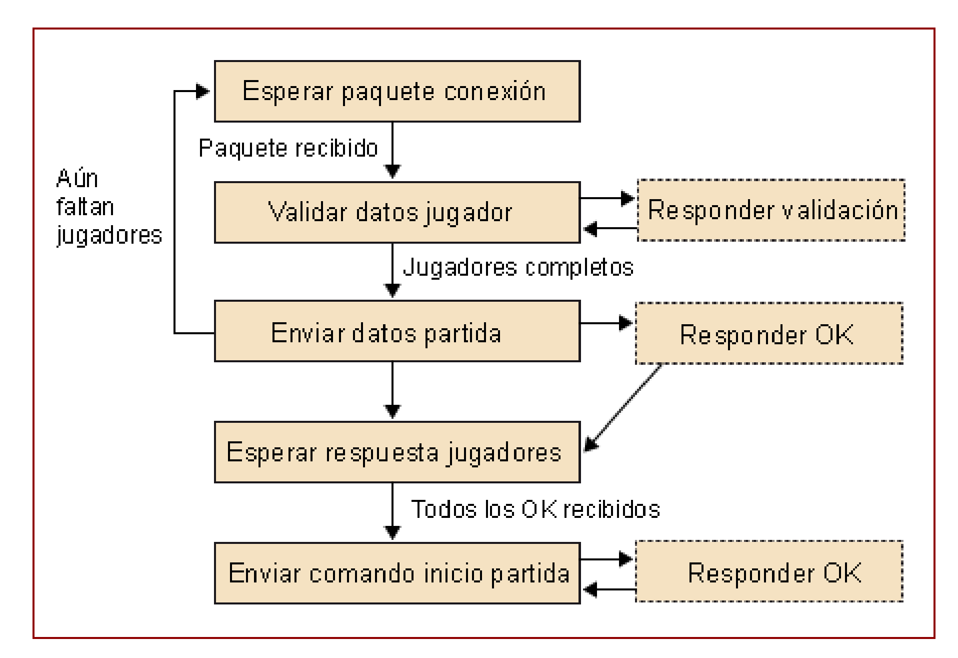



Figura 23. Ejemplo de los subestados del estado de conexión de los jugadores. Los cuadros discontinuos incluyen la participación de los clientes.

El diagrama más complejo acostumbra a ser siempre el del bucle principal de la partida. En este debemos incluir, aparte de los paquetes de datos que transmiten las acciones de los jugadores, otros paquetes de control que permitan pausar la partida, identificar si alguien ha desconectado, si hay un ganador, si ha habido un error en el servidor... Para cada uno de estos casos hay que

tener definido qué se va a hacer, qué paquetes se enviarán a los clientes y qué respuesta se espera de ellos, por lo que conviene detallarlos individualmente en el diagrama de estados.

Entraremos en más detalle en el diseño de los diferentes casos en el próximo módulo, cuando describamos los estados del juego y cómo se realizan las transiciones entre ellos.

## 3.6. Diseño de juegos en línea persistentes

Con la introducción de los juegos de red, apareció un nuevo tipo de juego en el que las partidas se desarrollan completamente en línea. Estos juegos tienen un servidor que está siempre funcionando (incluso cuando no hay jugadores), y los jugadores se conectan cuando quieren jugar una partida. La información del estado del juego se almacena siempre en el servidor, de manera que los usuarios pueden jugar desde cualquier ordenador usando su cuenta de usuario.

Pero quizás la característica más importante de estos juegos es que están pensados para permitir jugar simultáneamente a un gran número de jugadores (por ejemplo, se calcula que en el World of Warcraft pueden estar jugando millones de jugadores a la vez). Para mantener la infraestructura necesaria que permita jugar a todos ellos, algunos juegos utilizan una suscripción mensual que sirve para pagar los gastos de mantenimiento del servidor y del propio videojuego.

El requisito técnico de este nuevo género es disponer de dos nuevos perfiles dentro del desarrollo y la explotación de un videojuego que no se habían utilizado anteriormente:

- Un **administrador de red**. Debe encargarse de garantizar el funcionamiento óptimo de las comunicaciones.

- Un **administrador de bases de datos**. Tiene que garantizar el almacenamiento correcto de los datos, así como proporcionar un acceso rápido a ellos.

En este apartado vamos a analizar cuáles son las principales diferencias de un juego en línea persistente con un juego multiplayer normal. Comentaremos la arquitectura del juego y algunas técnicas de programación utilizadas.

### 3.6.1. Arquitectura del sistema

Vamos a discutir los elementos básicos de la arquitectura de un juego en línea. Este tipo de juegos necesita un diseño mucho más detallado de esta arquitectura por varias razones:

- El diseño es mucho más complejo, ya que se tiene que decidir qué partes se ejecutan en el cliente y qué partes se ejecutan en el servidor.

- El sistema tiene que estar preparado para soportar todos los jugadores que van a utilizar el juego simultáneamente.

- La cantidad de información que se transmite entre cliente y servidor es mucho mayor.

- Todos los datos se guardan en el servidor, así que nos tenemos que encargar de que estos datos se encuentren siempre disponibles.

En primer lugar, la infraestructura de red de un mundo persistente se compone de tres capas distintas:

- Los clientes que corren la interfaz gráfica para que puedan interactuar con el entorno.

- Los servidores que ejecutan el código del juego y deciden lo que está pasando en todo momento.

- Los servidores de bases de datos que guardan toda la información referente a los usuarios y algunos datos sobre el estado del juego.

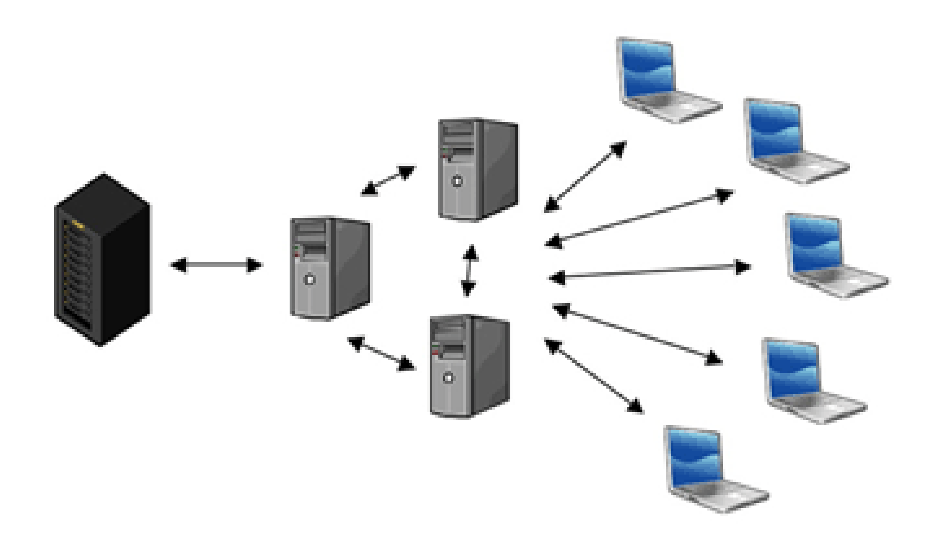


Figura 24. Esquema de una arquitectura para un juego en línea. A la izquierda tenemos un servidor de bases de datos con los datos del juego, en medio, los servidores donde se ejecuta el juego y, finalmente, a la derecha, están los clientes que ejecutan la interfaz gráfica para poder interactuar

**Bases de datos**

La base de datos juega un papel clave en guardar y proveer la información del juego. Algunos aspectos que cabe tener en cuenta cuando diseñamos esta parte de la arquitectura son:

- Es muy importante dimensionarla correctamente para garantizar que el acceso a los datos no sea un cuello de botella que ralentice el juego. Una posibilidad es utilizar varias bases de datos conectadas entre sí mediante un sistema de balanceo de carga, que distribuye las consultas de los servidores entre varias máquinas para mejorar la eficiencia del proceso.

- También es clave realizar copias de seguridad periódicas del contenido de la base de datos, ya que un problema en ésta afectaría directamente a todos los usuarios del juego.

La comunicación mediante los servidores y la base de datos normalmente se realiza mediante comandos SQL (*structured query language*), un lenguaje que permite decir a la base de datos qué es lo que nos interesa y poder obtener el resultado en un formato estándar. No vamos a entrar en más detalles sobre estos comandos, ya que caen fuera del contenido del curso, pero el lector puede encontrar infinidad de tutoriales y guías de referencias gratuitas en Internet.

También existe la posibilidad de guardar todos los datos en ficheros. El acceso a los datos es por tanto más rápido y además el sistema es mucho más fácil de implementar. Por otro lado, trabajar con ficheros es mucho menos escalable, tanto en el volumen de datos que se puede soportar como en la posibilidad de añadir nuevos campos en los datos.

**Arquitectura del cliente**

En este tipo de juegos, el cliente es parecido a lo que conocemos en informática por un "terminal tonto". Este tipo de terminales tan sólo sirve para recoger la entrada del usuario, enviarla al servidor y esperar la respuesta del servidor para mostrarla por pantalla.

**Ejemplo de cliente tonto**

Otro ejemplo de cliente "tonto" son los terminales de los primeros ordenadores Unix, los cuales sólo eran capaces de leer las teclas del teclado, enviarlo a un sistema central y devolver la información a la pantalla.

En el caso de un juego, esto nos da la posibilidad de dedicar todos los recursos del sistema para potenciar el entorno gráfico y sonoro, ya que no tenemos que dedicar la CPU para hacer cómputos de inteligencia artificial o de colisiones (de esto ya se encarga el servidor). Por tanto, podemos afirmar que uno de los aspectos más importantes a tener en cuenta con respecto al cliente, junto con la interfaz de red que nos comunica con el servidor, es el diseño de la interfaz de usuario.




Figura 25. Ejemplo de una interfaz de usuario del juego Everquest © Sony online entertainment

Si queremos mejorar la experiencia del cliente, existe la posibilidad de implementar pequeños trucos dentro del código del cliente que mejoren la eficiencia de la red. En el siguiente apartado os detallaremos algunas maneras de prevenir el retraso de los paquetes y hacer que el cliente siempre tenga la sensación de movimiento, aun cuando no estemos recibiendo información.

> Una de las grandes ventajas de separar la parte gráfica de la parte lógica es que podemos diseñar varios tipos de clientes que se ajusten a las capacidades hardware de los usuarios. Un ejemplo de la utilización de esta técnica se encuentra en el Ultima Online, donde los usuarios pueden escoger entre varias versiones de cliente que utilizan gráficos 2D o gráficos 3D.



Figura 26. Una misma escena vista en 2D y en 3D en el juego Ultima Online © Origin Systems / Electronic Arts

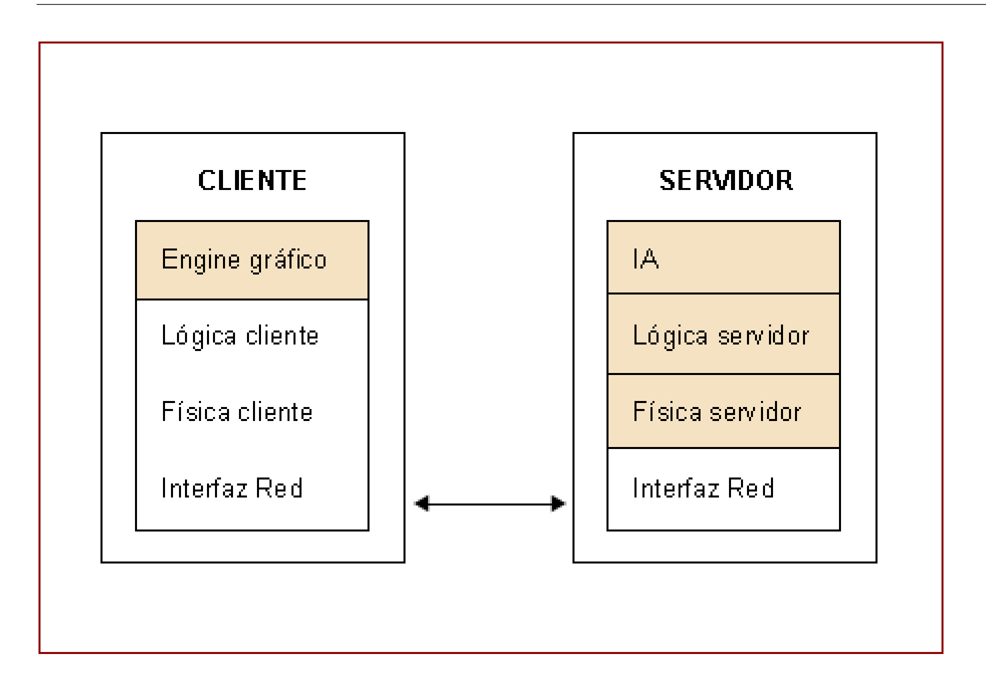
### Arquitectura del servidor

Por otra parte, el servidor está construido básicamente por **tres componentes**:

- Una interfaz de red que nos permita comunicarnos con el servidor de bases de datos.
- Una interfaz que nos permita comunicarnos con los clientes, recibir sus acciones y comunicarles lo que está ocurriendo a su alrededor. Este sistema tiene que estar muy bien optimizado, ya que la cantidad de datos recibidos/enviados es enorme.
- El elemento más importante es lo que llamamos un 'engine lógico'. Este engine se encarga de gestionar todas las entradas de los jugadores y del sistema de inteligencia artificial y de decidir qué está ocurriendo en cada paso.

El servidor no utiliza ningún tipo de información gráfica ni sonora; sólo precisa de aquella información necesaria para controlar qué están haciendo los jugadores (posiciones, acciones...).

En el siguiente esquema presentamos una distribución muy simple de repartición de los componentes principales entre el cliente y el servidor. La parte de lógica y física es mucho menos importante en el cliente; en estos dos módulos tan sólo se trata de pequeños trozos de código para ayudar a que el juego vaya más fluido (como predicciones de la posición de los jugadores), y evitar que pueda haber colisiones entre personajes que el servidor no nos ha enviado correctamente. El trabajo importante en estos dos puntos se realiza en la parte del servidor, donde se recogen todos los movimientos de todos los jugadores, se añaden todas las decisiones tomadas por el módulo de inteligencia artificial, se realiza la actualización del mundo y se reenvía el nuevo estado a todos los clientes.




### 3.6.2. Segmentación del mundo

Para garantizar que todos los usuarios puedan disfrutar de una buena experiencia de juego, es necesario limitar el número de conexiones máximas que el servidor puede gestionar. Por lo tanto, a veces es necesario utilizar varios servidores que garanticen que todos los clientes puedan conectarse y jugar al mismo tiempo.

Hay **dos técnicas** para distribuir usuarios entre servidores:

- La primera técnica es la de crear varios servidores a los cuales los clientes se pueden conectar, conocidos como reinos (*realms*). Cada uno de estos servidores contiene una copia completa del mundo, y no se puede transferir la información de un usuario de un mundo al otro. Algunos juegos como World of Warcraft aplican diferentes reglas a cada uno de estos servidores, lo que amplía las posibilidades multijugador del juego.
  Una ventaja adicional de estos servidores es que los podemos colocar en diferentes localizaciones, lo que permite reducir el tiempo que tardan los paquetes entre cliente y servidor, mejorando la fluidez del juego.




Figura 27. Pantalla de selección de servidor en World of Warcraft © Blizzard Entertainment

- La segunda opción es lo que se conoce por una subdivisón espacial del mundo. Este sistema es mucho más complejo. Consiste en recortar el mapa donde se desarrolla la acción en varias zonas y colocar la gestión de cada una de ellas en un servidor diferente.
  Se debe tener en cuenta que esta división no tiene por qué hacerse en tamaños equivalentes. Tendremos que estudiar la densidad de población en cada una de las zonas del mundo (es importante estudiar las ciudades y puntos de entrada de usuarios) y dividir para que el número de usuarios sea parecido en todas ellas.
  Normalmente, el salto entre dos zonas distintas se realiza mediante lo que se conoce como un "portal", que es la representación gráfica de un salto de una zona a la otra. Aun así, existen técnicas más profesionales que permiten segmentar el juego sin dar la impresión al usuario de que está realizando un cambio entre dos servidores distintos.

### 3.7. Técnicas avanzadas para juegos de red

En los últimos años se han desarrollado algunas técnicas complementarias para mejorar dos aspectos puntuales de los juegos en red, la seguridad de las conexiones y el rendimiento de la red. Ambas técnicas ayudan a que los usuarios no perciban algunos de los problemas que pueden existir en la conexión con los otros clientes.

### 3.7.1. Mejora del rendimiento de red

Uno de los problemas más comunes en las partidas de red es que los paquetes con información de lo que están haciendo los otros jugadores no llegan de forma constante. Puede haber todo tipo de problemas de comunicación: pequeños cortes, retrasos, paquetes que no llegan en orden...

El trabajo del programador es anticiparse a todos estos problemas y hacer que no sean visibles para el usuario. Vamos a comentar tres técnicas que se utilizan para llevar a cabo esta acción de forma transparente al usuario.

#### Mejorar la comunicación a nivel de red

La primera opción que tenemos para mejorar nuestro sistema es a nivel de hardware, sistema operativo y conexión hacia Internet. Es muy importante que nuestro sistema esté preparado para poder enviar y recibir todos los datos que necesitamos. Es importante que intentemos utilizar una conexión que nos permita mejorar algunos de estos aspectos:

- Reducir al mínimo el número de paquetes perdidos.

- Reducir al mínimo el tiempo de latencia, es decir, el tiempo que tarda en viajar un paquete entre el origen y el destino.

- Reducir al máximo el *jitter*, es decir, la diferencia entre el tiempo de llegada de dos paquetes consecutivos. Nos interesa mantener un ritmo constante para poder integrar el flujo de entrada/salida coherentemente en la aplicación.

**RTT**

Una de las medidas más utilizadas para conocer el tiempo de viaje de los paquetes es el RTT (*round trip time*), que mide el tiempo necesario para ir al destino y volver al cliente, ya que el protocolo TCP/IP no garantiza que los dos caminos pasen por el mismo sitio.

Estas mejoras se realizan normalmente a base de invertir más dinero en infraestructura, ya que no se pueden realizar mediante programación.

#### Optimización del protocolo y de la secuencia de conexión

Otro elemento clave para mejorar la fluidez de un juego es la optimización del protocolo de comunicación. Vamos a dar **algunos consejos** sobre cómo puede mejorarse el tamaño de los paquetes y la cantidad de paquetes que se envían:

- Enviar únicamente los cambios de estado en lugar de enviar continuamente el estado actual. Por ejemplo, en lugar de enviar nuestra posición continuamente, sólo enviamos la orden de avanzar, retroceder.... Este proceso requiere que cada cierto tiempo se envíe algún paquete de sincronización para garantizar que ambos sistemas están viendo el mismo estado.

- Elegir un área alrededor del personaje y enviar únicamente aquellos eventos que ocurran dentro de esta zona. También podemos establecer diferen-

tes secciones más grandes en las que podemos enviar menos información (por ejemplo, sólo posiciones de personajes para dibujar el mapa).

- Intentar condensar el máximo de información en un solo paquete de datos. Más paquetes implica un mayor *overhead* (ya que enviamos más cabeceras), y por tanto generamos más tráfico de red del que es necesario.

- Mensajera jerárquica: se trata de adaptar la cantidad de información enviada según las diferentes velocidades de los clientes (podemos tener desde líneas RDSI, hasta conexiones por cable o por ADSL con todo tipo de velocidades). La técnica consiste en asignar una cierta prioridad a los mensajes que vamos a enviar y decidir si lo transmitimos o no dependiendo de la importancia que tenga para el correcto funcionamiento del juego. Por ejemplo, los cambios de posición tienen una prioridad muy alta, pero un cambio en la animación del usuario es de prioridad muy baja.

**Predicción de los movimientos de los jugadores**

Otro de los métodos más usados para que el usuario no note los problemas que pueda haber en la red es el uso de la predicción del comportamiento de los oponentes.

El cliente siempre tiene que estar intentando saber cuál es el próximo movimiento que van a realizar los objetos que se encuentran en su zona. Cada vez que recibe del servidor una posición y una orientación, simplemente hemos de corregir parcialmente la posición de los objetos que habíamos intuido para que se ajusten a la realidad. Podemos utilizar algoritmos que hagan esta transición de forma simple, para que no veamos cambios bruscos en las posiciones de los elementos.

Un buen algoritmo de predicción necesita dos componentes principales. Los datos históricos de que han hecho todos los elementos en los últimos informes del servidor y un buen sistema de predicción que permita anticipar el movimiento real de los objetos. Aun así, no hay ningún sistema que sea capaz de avanzarse siempre a las acciones de los otros usuarios y los elementos del sistema, por lo que tenemos que tener claro que el resultado de nuestra predicción no es nunca fiable al 100%.

En algunos juegos en línea, la predicción da lugar a situaciones bastante extrañas o cómicas. Aunque la sensación de que estamos jugando es bastante continua, si ha habido un corte en la red puede ser que aparezcamos de repente en un punto donde nos encontrábamos hace un buen rato. Esto es debido a que cuando se recupera la conexión vemos que existe una gran diferencia entre el estado del cliente y el del servidor, con lo que el servidor nos vuelve a situar en la ultima posición que él conocía como buena.

En algunos juegos este problema puede ser más preocupante, ya que si la conexión se corta en un momento clave (en un combate, por ejemplo), cuando el servidor nos actualiza el estado podemos descubrir que nuestro personaje ha muerto. En estos casos de poco nos sirve la predicción de las acciones.

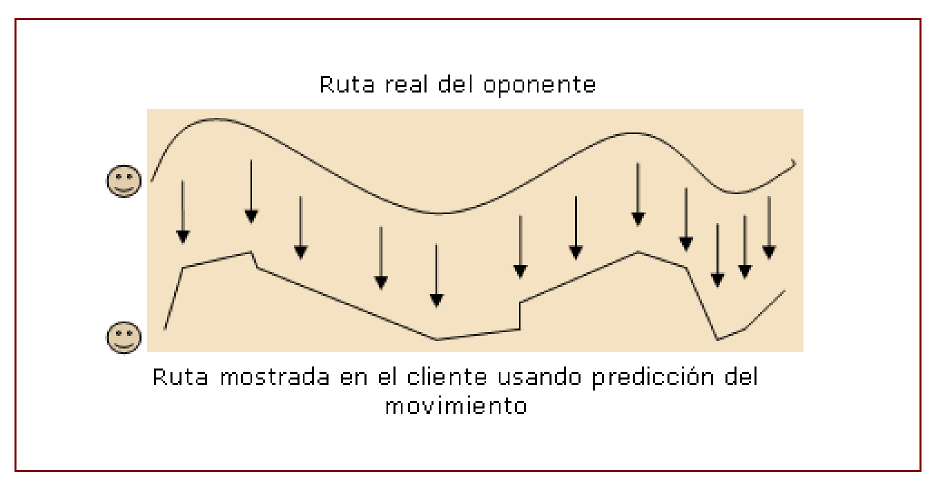



Figura 28. Ejemplo de una ruta real y la misma ruta predicha por el usuario. Cada vez que recibimos una actualización de la posición, corregimos el camino para que se adapte a la situación real.

### Envío de acciones en lugar de posiciones/estado

En algunos tipos de juegos, sobre todo los que requieren mucho movimiento de elementos (como los RTS), no es viable enviar en cada momento las posiciones y el estado de todos los elementos, ya que esto conllevaría un gasto de ancho de banda considerable. En su lugar se envían las ordenes que dan los jugadores y a qué elementos del juego se las dan.

Para que este sistema funcione todos los clientes tienen que ejecutar la misma versión del juego, a la misma velocidad de ejecución (se acostumbra a trabajar a 10fps) y este debe ser totalmente determinista. Es decir, si un jugador da una orden a un comando de mover 100 unidades de A a B, todos los clientes deben calcular la misma ruta para los 100 y empezar a moverlos al mismo momento.

El elemento clave en el sistema es la sincronización inherente. Aunque no se envían mensajes, si el juego está bien implementado, se supone que todos los sistemas deben estar sincronizados en todos los ciclos de ejecución. Cada cierto tiempo se debe llevar un testeo de sincronización entre todos los elementos del juego (por ejemplo, comprobar que todas las unidades están en la misma posición y/o en el mismo estado). Un solo fallo de sincronización no tiene solución y conllevará la finalización de la partida.

**Garantizar el determinismo**

Uno de los problemas más importantes que hay que solucionar para garantizar el determinismo es la resolución de los números de coma flotante.

### 3.7.2. Seguridad de las comunicaciones

La seguridad en la comunicación es necesaria para garantizar el correcto desarrollo de un juego. La seguridad es importante por tres razones principales:

- Para proteger la información sensitiva de los usuarios (contraseñas...).
- Para evitar suplantaciones de personalidad (un usuario se hace pasar por otro).
- Para proporcionar un entorno de juego justo, donde ningún usuario pueda aventajar a los otros utilizando trucos o trampas.

La protección de la conexión se puede realizar mediante diversas técnicas que actúan en diferentes niveles de la arquitectura TCP/IP:

- En cuanto a la red o al transporte, podemos filtrar paquetes extraños o maliciosos. En este ámbito podemos detectar y prevenir algunos de los ataques clásicos de los *hackers* que intentan modificar direcciones y puertos para engañar a los servidores.

- En cuanto a la aplicación, podemos proteger los datos cifrando la información contenida en los paquetes. Sólo es recomendable aplicar esta técnica en momentos puntuales y cuando se trate de información sensible, ya que el proceso de cifrado y descifrado requiere cierta computación que durante el desarrollo de una partida no nos podemos permitir.

Otra alternativa es utilizar un software externo para gestionar la seguridad de nuestra aplicación. Servicios como PunkBuster proporcionan mecanismos para mejorar la seguridad de los juegos, y evitar que los usuarios utilicen trampas para sacar ventaja respecto a otros usuarios.

Al instalar el juego, instalamos también una pequeña aplicación que corre en paralelo al juego y que detecta cualquier intento de ataque o trampa. Adicionalmente, este tipo de servicios mantienen una base de datos sobre usuarios "baneados" (usuarios a los que se ha descubierto intentando realizar alguna acción ilegal), a los que ya no se permite acceder al juego.




Figura 29. PunkBuster, servicio que mantiene un control en los usuarios

## 3.8. Herramientas adicionales

Para finalizar este apartado, comentaremos algunas herramientas adicionales que se utilizan en los juegos de red. Se desarrollan para realizar tareas muy concretas que son independientes del funcionamiento normal del juego, pero que a su vez son necesarias para facilitar su funcionamiento a largo plazo.

### 3.8.1. Autopatchers

Se trata de un componente que permite actualizar el sistema de manera transparente para el usuario. Esto permite a los desarrolladores seguir mejorando el juego una vez ha sido publicado, corregir errores e introducir nuevas funcionalidades.

La actualización normalmente se puede efectuar antes de entrar en la aplicación (si se tiene que cambiar parte del código principal del programa) o durante el funcionamiento del juego (si sólo es necesario descargar recursos, como por ejemplo mapas, texturas...).

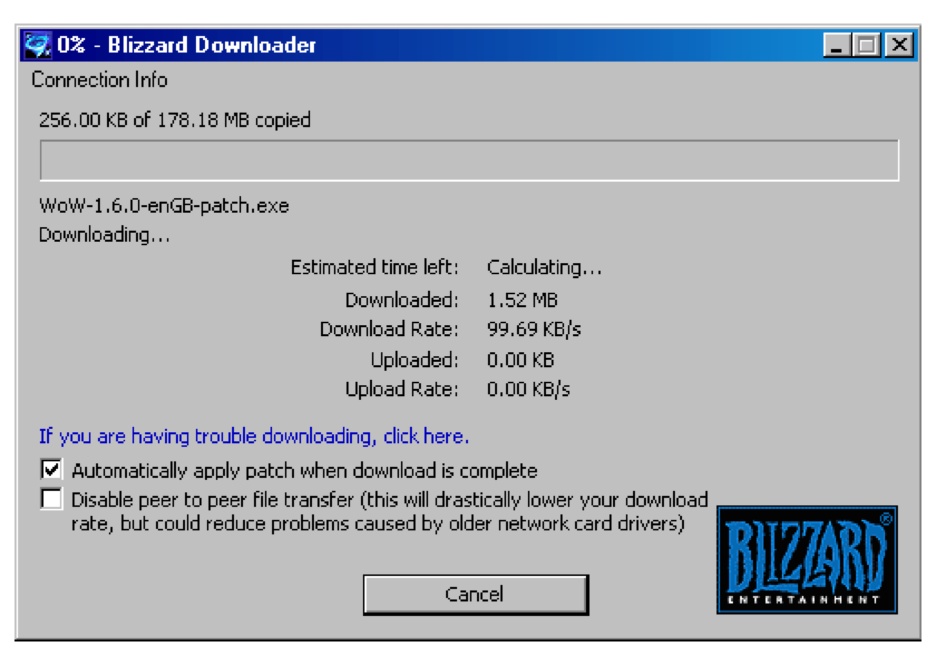



Figura 30. Ejemplo del sistema de autoactualización de World of Warcraft © Blizzar Entertainment

La librería RakNet utilizada en este capítulo proporciona un conjunto de clases que nos permiten desarrollar nuestro propio *autopatcher*.

### 3.8.2. *Master servers*

Otra utilidad muy importante para facilitar el juego en red es lo que se conoce por un *master server*. Un *master server* es un servidor que únicamente gestiona diferentes partidas de un juego. Los jugadores se conectan con un cliente incorporado en el propio juego, y allí tienen todas las partidas para escoger contra quién jugar. Adicionalmente, estos servidores proporcionan también salas de chat para encontrar otros jugadores.




Figura 31. Ejemplo de una conexión a un Master Server de Unreal Tournament © Epic Games




Figura 32. Ejemplo de una sala de chat en el Master Server de Diablo II © Blizzard Entertainment

La API de Raknet proporciona una clase para poder implementar un *master server* y gestionar todas las partidas que se estén realizando de un juego. También permite a los usuarios poder acceder a cualquiera de estas partidas

### 3.8.3. Sistemas de comunicación avanzados

Aunque la mayoría de juegos actuales permiten algún tipo de conversación dentro del propio juego, existen todo tipo de herramientas adicionales que facilitan que los diferentes jugadores hablen entre ellos.

La mayoría de estos complementos auxiliares proporcionan funcionalidades extendidas que no se encuentran por defecto en los juegos, como por ejemplo la comunicación por voz. En el caso de RakNet, ya hemos comentado que se puede integrar con las librerías FMOD para programar un chat por voz.

**Documentación**

En los tutoriales de la página web de la librería se explica cómo se puede desarrollar un cliente de chat.

# Resumen

En este módulo hemos estudiado los tres tipos de interacción que existen en un videojuego:

- Usuario a videojuego, donde hemos tratado todos los dispositivos de entrada y trabajado con la librería OIS.

- Videojuego a usuario, donde hemos defendido la creación de interfaces sencillas, fáciles y útiles estudiando conceptos como la metáfora y la usabilidad. También hemos trabajado en el aspecto teórico y práctico del sonido y, en los ejemplos, hemos usado la librería FMOD.

- Videojuego a videojuego, donde hemos presentado los diferentes protocolos de comunicación que existen para conectar dos o más videojuegos y lograr que varios usuarios jueguen en la misma partida desde ordenadores separados. Hemos estudiado la librería RakNet, con la que podemos crear con poco esfuerzo aplicaciones comerciales que trabajen en red.

# Actividades

**1.** Implementad un código usando OpenGL y OIS, donde se dibuje un mundo 2D basado en *tiles*. A partir de los eventos del ratón, el usuario deberá seleccionar y mover los diferentes *tiles* que conforman el mundo. Reservad en la parte izquierda una zona donde se muestren los diferentes tipos de *tiles* para que el usuario pueda diseñar visualmente un mundo 2D. Intentad que la interfaz gráfica sea sencilla y fácil de usar.

**2.** Inventad una serie de *tiles* que emitan sonido (cascadas de agua, fuego, una atracción de feria...) e incorporadlos en el editor anterior. Cread otra aplicación que permita al usuario moverse por el mundo creado con el editor (se deberá guardar en un fichero) y, utilizando la librería FMOD, haced que el usuario oiga correctamente todos los sonidos. El usuario deberá moverse usando las teclas y con el *joystick* (si tenéis).

**3.** Cread un sistema para que dos instancias de la aplicación puedan compartir la información de los avatares que hay en el mundo usando la librería RakNet. De esta manera, dos jugadores pueden interactuar en el mismo mundo. Haced que un jugador pueda oír un sonido característico del otro jugador (un silbido, por ejemplo).

# Glosario

**API** *m* Application Programming Interface. Es una interfaz proporcionada por una librería para poder acceder a sus funciones.

**banear** *v tr* No permitir el acceso a un juego por haber realizado trampas anteriormente.

**cheat** *m* Se trata de algún tipo de truco o trampa utilizado para mejorar de forma no legal en el juego.

**cliente/servidor** *m* Arquitectura de comunicación en la que un servidor central actúa como coordinador de la comunicación. Todos los mensajes deben pasar por él antes de llegar al destino.

***force feedback*** *m* Sistema para introducir una fuerza de reacción a las acciones del usuario para proporcionarle una sensación de juego más realista.

**HUD** *f Head up display*. Metáfora que utilizamos para mostrar información adicional al usuario del estado de la partida: salud, balas, recursos...

**máscara de selección** *f* Sistema para poder decidir qué elementos del juego ha escogido el usuario para realizar una cierta acción.

**metáfora** *f* Interfaz visual que nos sirve para definir los comportamientos de elementos que componen nuestra aplicación.

**MMG** *m Massively multiplayer games*. Se trata de un tipo de juegos que sólo se puede jugar a través de la red. Nos conectamos a un servidor central que siempre está funcionando y allí jugamos nuestras partidas.

**MMORPG** *m massively multiplayer online role playing games*. Un grupo concreto de los juegos MMG centrado en los juegos de rol.

***peer-to-peer*** **(P2P)** *m* Arquitectura de comunicación donde todos los elementos actúan como iguales. La información se distribuye entre todos los clientes a la vez.

**protocolo** *m* Lenguaje de comunicación utilizado entre dos dispositivos.

**sistema *buffered/unbuffered*** *m* En un sistema de entrada *buffered* tenemos una tabla intermedia que nos informa del estado del dispositivo de entrada; en cambio, en un *unbuffered* debemos gestionar nosotros mismos el estado de los dispositivos.

***socket*** *f* Tecnología que permite abrir una conexión entre dos ordenadores y comunicarse a través de ella.

**TCP/IP** *Transmission control protocol / internet protocol*. Conjunto de protocolos que se utiliza para comunicar dos dispositivos a través de Internet.

***tracker*** *f* Una aplicación donde introducimos una serie de muestras (*samples*) o sonidos cortos que trataremos como instrumentos, y los utilizamos para crear una melodía.

# Bibliografía

**Alexandre, Thor** (2003). *Massively Multiplayer Game Development*. Hingham, MA: Ed. Charles River Media (2 vols.).

**Armitage, Grenville; Claypool, Mark; Branch, Philip** (2006). *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. Hoboken, NJ: John Wiley & Sons, Ltd.

**Barron, Todd** (2001). *Multiplayer Game Programming*. Boston, MA: Thomson Course Technology.

**Boer, James R.** (2003). *Game Audio Programming*. Hingham, MA: Charles River Media.

**Fox, Brent** (2004). *Game Interface Design*. Boston, MA: Thomson Course Technology.

**McCuskey, Mason** (2003). *Beginning Game Audio Programming*. Boston, MA: Muska & Lipman/Premier-Trade.

**Sanchez-Crespo Dalmau, Daniel** (2003). *Core Techniques and Algorithms in Game Programming*. Indianapolis, IN: New Riders.

**Young, Vaughan** (2005). *Programming a Multiplayer FPS In DirectX*. Hingham, MA: Charles River Media.